Synthesizing Verifiable Code for Large Specifications Using Few-Shot Learning

Saarland University

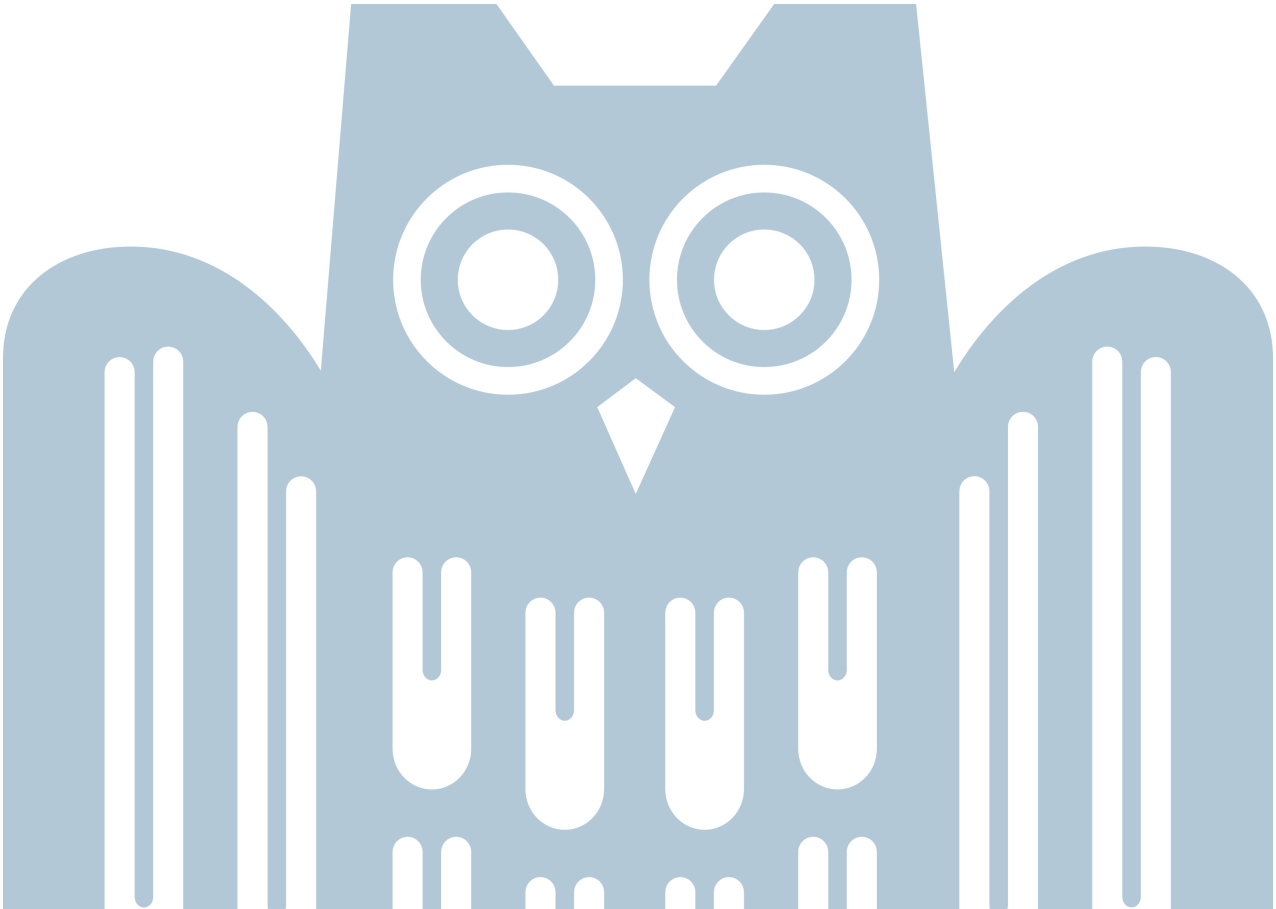
Department of Computer Science

Bachelor's Thesis

submitted by

Benedict Böttger

Saarbrücken, October 2023



Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Frederik Schmitt, M.Sc.
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Dr. Rebekka Burkholz
Submission:	October 2, 2023

Abstract

In this work, we present a novel approach which leverages the few-shot learning capabilities of Large Language Models (LLMs) to address the long-standing challenge of synthesizing digital circuits from linear-time temporal logic (LTL) specifications. We concentrate on parameterized specifications in this work, which are common in hardware specifications, as they can describe a system of arbitrary size dependent on one or more parameter values. Owing to the computational complexity (2EXPTIME-complete), classic synthesis algorithms are not able to scale beyond relatively small parameter values.

Using implementations that fulfill the specification for lower parameter values, we will task different LLMs with generating a satisfying implementation for a larger parameter value. The smaller implementations can be either generated by classical synthesis tools, which perform well on smaller specifications, or by using human-written solutions, in a hybrid approach. For the hardware representation we directly target the Verilog hardware description language instead of a more low-level representation like AIGER, in order to leverage the expressiveness of the language in combination with LLMs.

We demonstrate that this approach can, in some cases, successfully synthesize correct Verilog code using the examples given as well the LTL formula. This works especially well if based on human-written solutions. In the successful instances, this approach can scale to several orders of magnitude beyond what classical LTL synthesis tools like Strix or BoSy can do on their own. Intriguingly, in a limited number of cases, the LLM is able to accomplish this task zero-shot, not requiring any example implementations. While this method is not consistent enough to be used on its own, we think that a hybrid approach may prove useful, by integrating it with existing tools or a programmers workflow.

Acknowledgements

I would like to genuinely thank my advisor for his constant support with writing my thesis, always taking however much time was needed to answer my questions. I also want to express my gratitude to Prof. Finkbeiner. Listening to his lecture in my first year was on the moments which really helped to make me want to go deeper into computer science. During this thesis, he always had an open ear and never shyed away from giving me constructive feedback. To my parents and friends, who supported me unconditionally during writing my thesis as well as my whole studies. Finally I would like to thank Dr. Rebekka Burkholz for acting as a second reviewer for my thesis.

Eidesstattliche Erklärung
Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 2 October, 2023

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, 2 October, 2023

Contents

Chapter 1

Introduction

Linear-Time Temporal Logic (LTL) specifications [1] are often used to describe the temporal behavior of digital circuits. As such, they are commonly utilized for verifying correct behavior of low-level systems like FPGAs or other computer chips. This process is called *model checking* [2]. However, it requires building the system as well as the specification separately, both complex tasks. This essentially amounts to doing the same work twice, as the specification itself already contains all the information necessary to build the system. For that reason it has long since been proposed to directly synthesize an implementation from the specification alone [3]. The *synthesis problem*[1] is intriguing, since an efficient solution holds the promise of easing the development workflow to just having to maintain a correct specification. But even though model checking has been seeing wide use in industry for some time now, synthesis seldom makes an appearance. This mostly comes down to the computational complexity of the problem. Regarding LTL for example, the computational complexity of the LTL synthesis problem is exponentially greater than what is required for LTL model checking [4] (which already is computationally expensive). Restricting the types of specification has proven useful for this in the past. Piterman, Pnueli, and Sa'ar [5], for example, restricted the specifications to a specific subset that has a much lower theoretical complexity.

We, on the other hand, are only going to consider parameterized LTL specification. Note that this isn't a restriction in the sense that it reduces the theoretical complexity. It is instead a practical consideration. Parameterized specifications are typical in hardware applications and enable us to specify systems of an arbitrary size, dependent on one or more parameter values. One could, for example, describe an adder circuit for an

[1] Different specification languages exist, but for the purposes of this work, we will only cover LTL specifications and their corresponding model checking and synthesis problems

arbitrary number of bits n, the parameter in this case[2]. While classical tools perform well on smaller parameter values for those specifications, as soon as the parameter value is increased past a certain point, this will not be possible anymore. This is where we will be applying the pattern recognition power of large language models (LLMs) to try and go from these smaller examples to generalizing to larger specifications.

Our aim is to leverage the patterns in the LTL specifications that occur when scaling the parameter values, which can carry over to the implementations, depending on how they are generated. This is motivated by the success seen in recent years in utilizing machine learning generally and LLMs specifically for logical reasoning tasks (see Sect. 2.5) and why we are exploring the application of using LLMs for LTL synthesis. Central to our work is the question of whether it is possible for an LLM, given some examples for smaller parameter values, to pick up on the patterns in the code and generate a solution for larger specifications, for which classical tools will time out. This technique of giving some reference examples before the actual question is called *few-shot learning* and has seen a lot of success recently [6, 7, 8, 9, 10]. For the actual hardware representation we are using the Verilog hardware description language (HDL) [11], which has a reasonably high-level syntax allowing for relatively compact modules. For the purposes of synthesizing code we are especially hoping that the expressiveness of Verilog will give an advantage compared to more low-level representations like AIGER [12].

→ Sec. 2.5, p. 11

In this work, we will first present the relevant background to our work (Chapter 2), encompassing both formal methods as well as machine learning and LLMs. Afterwards we will go over how we set up our experiments (Chapter 3), present detailed results (Chapter 4). We will finally close with a summary, showing how our method can greatly extend what state-of-the-art synthesis tools can do on their own, as well as giving an outlook for possible future work (Chapter 5).

→ Chapter 2, p. 3
→ Chapter 3, p. 13
→ Chapter 4, p. 19
→ Chapter 5, p. 29

[2] An adder circuit doesn't possess temporal properties but will be more familiar to most readers. A better example to also show the temporal restrictions would be an arbiter circuit which sequentially grants request one at a time.

Chapter 2

Background

2.1. Linear Temporal Logic (LTL)

LTL was first introduced by Amir Pnueli [1] as a way to formally describe the behavior of computer programs. It extends propositional logic by the temporal operators $\bigcirc$ (Next) and $\mathcal{U}$ (Until). Starting relative from the current time step, $\bigcirc$ describes the following time step, and $a \,\mathcal{U}\, b$ requires a to hold until b becomes true. Further operators can be derived from these.

Formally, the syntax of any *LTL formula* φ is defined by the following grammar:

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

Where p comes from a finite set of atomic propositions AP.

We define the semantics of LTL as a language of infinite words, as described by Baier and Katoen [2]. An LTL formula φ defines a language over the alphabet 2^{AP}:

$$\mathcal{L}(\varphi) = \{\delta \in (2^{AP})^\omega = A_0 A_1 A_2 \ldots \mid \delta \models \varphi\}$$

with $\models$ being the smallest relation satisfying:

$$\begin{aligned}
&\sigma \models \text{true} \\
&\sigma \models a && \text{iff} \quad a \in A_0 \\
&\sigma \models \varphi_1 \wedge \varphi_2 && \text{iff} \quad \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \\
&\sigma \models \neg\varphi && \text{iff} \quad \sigma \not\models \varphi \\
&\sigma \models \bigcirc\varphi && \text{iff} \quad \sigma[1\ldots] = A_1 A_2 A_3 \ldots \models \varphi \\
&\sigma \models \varphi_1 \,\mathcal{U}\, \varphi_2 && \text{iff} \quad \exists j \geqslant 0.\ \sigma[j\ldots] \models \varphi_2 \quad \text{and} \quad \forall 0 \leqslant i < j.\ \sigma[i\ldots] \models \varphi_1
\end{aligned}$$

In practice we also often use the derived operators $\square$ (Forever) and $\Diamond$ (Eventually). $\square$ describes a property that has to hold true in every time step (from the current time step) while $\Diamond$ expresses that in some future time step (including the current one), a property must hold true. They are formally defined as:

$$\Diamond\varphi \quad \equiv \quad \text{true}\, \mathcal{U}\, \varphi \qquad\qquad \square\varphi \quad \equiv \quad \neg\Diamond\neg\varphi$$

As an example we will define a simple specification for a traffic light (controller), ensuring the following two properties. 1) The red and green lights should never be on at the same time and 2) in order to ensure traffic flow, there should always be red as well as green phases. This could be achieved by the formula $\square((\neg r \vee \neg g) \wedge \Diamond r \wedge \Diamond g)$ where r and g correspond to the red and green light.

The addition of temporal operators has an interesting effect on the solutions. While solving a formula in propositional logic only requires finding a single satisfying assignment, LTL requires finding a satisfying assignment each for an infinite amount of discrete time steps, called an *infinite trace*.

2.1.1. LTL Specifications

LTL formulas are commonly used to model the behavior of digital circuits, where each input and output signal corresponds to an atomic proposition whose value can change in each time step. Verifying one of these circuits presents a challenge though, as only the outputs can be controlled. To represent this notion, we formally define an *LTL specification* as a triple (φ, I, O) consisting of an LTL formula together with the two sets I (inputs) and O (outputs) which partition the set of atomic propositions $AP = I \dot\cup O$. In order to satisfy a formula φ, any circuit must respond to the inputs and set its outputs accordingly. This can also be seen in Fig. 2.1, where a detector circuit is described, where the output g is only true if and only if all of the inputs r_0, r_1, r_2, r_3 have been true since the last time.

→ Fig. 2.1, p. 4

$$\varphi = (\square(\Diamond r_0) \wedge \square(\Diamond r_1) \wedge \square(\Diamond r_2) \wedge \square(\Diamond r_3)) \leftrightarrow \square(\Diamond g)$$

$$I = \{r_0, r_1, r_2, r_3\}$$

$$O = \{g\}$$

Figure 2.1.: The LTL specification for a 4-bit input monitor ("detector"), which outputs a signal once all input signals have been high and then resets.

In practice, we are often going to use the format TLSF [13] to express our LTL specifications. Not only can it easily be converted into other formats using the tool

SyfCo [13], it also lets us use some syntactical sugar to make writing LTL specifications more manageable. It critically also allows us to express parameterized LTL, which makes it possible to describe systems at different scales by changing a single parameter. For example, the specification in Fig. 2.1 can be generalized to $(\bigwedge_{0 \leqslant i < n} \square\lozenge r_i) \leftrightarrow \square\lozenge g$ → Fig. 2.1, p. 4 (with $I = \{r_i \mid 0 \leqslant i < n\}$, $O = \{g\}$). These parameterized LTL specifications can later be expanded into regular LTL specifications by fixing the parameter values. Setting $n = 4$ here would result in the exact same specification as in Fig. 2.1. The exact syntax of → Fig. 2.1, p. 4 TLSF and how to express parameterized specifications in it are described in their format description [13]. Fig. 2.2 provides an example for the aforementioned specification → Fig. 2.2, p. 5 expressed in TLSF.

```
INFO {
  TITLE:       "Parameterized Input Monitor"
  DESCRIPTION: "Checks whether all input signals are eventually true"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
GLOBAL {
  PARAMETERS {
    n = 4;
  }
}
MAIN {
  INPUTS {
    r[n]; // request signals
  }
  OUTPUTS {
    g; // grant signal
  }
  GUARANTEES {
    &&[0 <= i < n] G F r[i] <-> G F g;
  }
}
```

Figure 2.2.: The specification for a n-bit parameterized input monitor (“detector”) in TLSF format. The LTL formula is specified in ASCII characters: $G = \square$, $F = \lozenge$

2.2. Reactive Synthesis

The problem of synthesizing a program from a formal specification alone was first stated in 1957 by Church [3]. Since then many approaches have been put forth in an attempt to efficiently solve this problem, some of which we will specifically discuss in Sect. 2.2.1. → Sec. 2.2.1, p. 6 The alluring promise is that instead of having to maintain complex code, it is enough to

write just the required specifications. The result is that the synthesized code is provably correct w.r.t. the specification, making this a powerful technique especially for use in safety critical systems. A simple example, again, is a traffic light controller, where you want to ensure that e.g. the red and green lights shouldn't be on at the same time.

Reactive synthesis specifically refers to the problem of synthesizing a *reactive module* (program) such that it satisfies a formal specification [4]. A reactive module is a common model for hardware circuits, representing a continuously running system which has a fixed set of inputs and outputs. Since the inputs can change in each discrete time step, the state and thus output can be dependent on multiple inputs seen over time. In order to specify the behavior of these reactive modules, LTL is commonly used as a specification language, though other languages exist as well. In this work, we will focus on LTL synthesis only.

2.2.1. LTL Synthesis

In practice though, it is hard to synthesize code from an LTL specification. In fact, it can be proven that the LTL synthesis problem is 2EXPTIME-complete [4] (solvable by a Turing machine in $O(2^{2^{n^k}})$ time steps for some fixed $k \in \mathbb{N}$ where $n \in \mathbb{N}$ is the input size), which makes solving the general case infeasible for larger specifications. In comparison, the act of verifying the correctness of a specification against a problem is called *model checking* and is in itself PSPACE-complete [2] (which is a subset of EXPTIME), so it usually can be performed much faster.

We will now present two widely used LTL synthesis tools, which use distinct approaches, namely *bounded synthesis* and *game-based synthesis*. These tools, both state-of-the-art in their respective approach, will provide a baseline to our work and is also what we will be comparing our results against.

BoSy [14, 15] uses an approach called *bounded synthesis*. In contrast to other approaches, it always produces a minimal solution (in the number of states). It works by first translating the formula into a constraint system. It then sets a bound on the number of states of the implementation and then tries to solve the system using an off-the-shelf constraint solver. We are using the default, which is RAReQS [16]. If no solution can be found, the bound is iteratively increased until a solution is reached, or the specification is found to be unrealizable (by solving for the negation of the formula) [17]. Note that even though the state space is technically finite for LTL synthesis, the exponential blowup manifests quickly. From initial observations it also seems like the solutions BoSy provides scale somewhat predictably.

Strix [18, 19], on the other hand, relies on a technique that goes much further back: *game-based synthesis*, originally described by Büchi and Landweber [20]. For LTL, game-based synthesis translates the LTL formula into a two-player (usually parity) game between the system and environment. In each round, the environment player chooses the value of the input variables, after which the system player tries to set the output

variables such that it satisfies the specification. If a winning strategy can be found, the specification is realizable. The strategy can then be translated into e.g. an AIGER implementation. Strix specifically uses parity games and employs nondeterministic techniques in solving them. This results in Strix sometimes producing different solutions on separate runs, which makes the scaling behavior harder to predict. The solutions are also not necessarily minimal.

At the time of writing, Strix is the state-of-the-art LTL synthesis tool, as measured by the SYNTCOMP competition [21, 22, 23].

2.3. Hardware Representations

2.3.1. AIGER And-Inverter Graph

An *And-Inverter Graph* (AIG) is a type of circuit only consisting of AND gates as well as NOT gates, allowing it to describe any combinational (stateless) circuit [24]. In what is commonly knows as a Sequential AIG, the representation is extended by a memory element, e.g. a D-flip-flop or latch [25]. This facilitates model checking due to the reduced complexity, without sacrificing expressiveness [26].

A common way to store digital circuits as Sequential AIGs is by using the AIGER And-Inverter Graph Format, which is a file format originally designed as a way to offer a concise file format for use in model checking competitions [12, 27]. It is often used as an output format by LTL synthesis tools such as Strix or BoSy. Additionally it allows to encode LTL properties in the AIG which can then be verified by a model checking tool like nuXmv [28].

```
aag 6 3 0 1 3
2
4
6
13
8 6 2
10 4 3
12 11 9
i0 select_0
i1 in_0
i2 in_1
o0 out
```

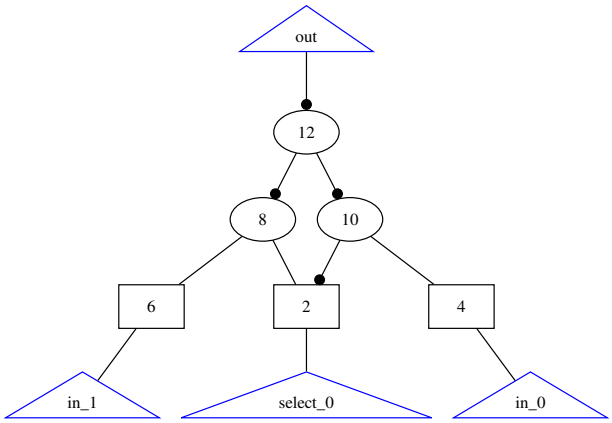


Figure 2.3.: An exemplary AIGER circuit and its corresponding graph representation

2.3.2. Verilog

Verilog [11] is a *Hardware Description Language* (HDL) used to design digital circuits at an abstraction level close to the actual hardware. Even though the syntax might look somewhat familiar at first glance (Fig. 2.4), the logic is fundamentally different from higher-level programming languages. Instead of operating on variables, it directly operates on signals and registers. Signals (carried by e.g. a wire), can be in a high or low state (binary, i.e. a single bit). Operations are usually performed in synchronization with a clock signal. Instead of using functions, code is encapsulated in *modules*, which can then be reused. An exemplary module implementing the LTL specification laid out in Fig. 2.1 is shown in Fig. 2.4. As Verilog is widely used in both industry and academia, it provides us with mature preexisting tools to build upon. We can convert Verilog code to AIGER (for model checking) using the open-source tool Yosys [29]. It is also possible to apply optimizations and transformations at the same time, which can reduce the size of the resulting circuit. However due to our reliance on Yosys for this step, we are limited to the Verilog standard supported by Yosys, which is largely identical to the Verilog-2005 standard [11]. It additionally supports some features from SystemVerilog [30] (an extension to Verilog) which are useful for verification, like the `$global_clock` variable.

→ Fig. 2.4, p. 8
→ Fig. 2.1, p. 4
→ Fig. 2.4, p. 8

```
module detector (
  input [3:0] r, // 4-bit input wire
  input clk, // clock input
  output reg g //output register
);
  reg [3:0] state; // a 4-bit register
  initial state = '0; // initialize the state to all zeroes
  always @(posedge clk) begin // perform this on every clock cycle
    state = state | r;
    g = 0;
    if(state == '1) begin
      g = 1;
      state = '0;
    end
  end
endmodule
```

Figure 2.4.: An exemplary Verilog module

2.4. Large Language Models (LLMs)

The problem of natural language processing (NLP) has been a long standing one. This includes tasks like classification, translation, text completion. A lot of different approaches and models were proposed over the years. Early examples include n-gram models [31] and rule-based approaches [32]. Later, Recurrent Neural Networks like LSTMs [33] were a popular choice for different kinds of sequence processing, including NLP.

But in 2017, a revolution in the field was kickstarted by [34] with the introduction of the *Transformer* model. Recurrent Neural Networks are notoriously difficult to train due to their high depth which often leads to vanishing and exploding gradients [35]. Transformers mostly solve this due to their unique self-attention mechanism, which can capture relationships between different tokens, usually within a specified context window which is limited due to practical considerations (in the standard Transformer architecture, there is a quadratic scale-up in computational expenditure in regards to the size of the context window). The discovery of the transformer sparked the development of numerous model variations, which constantly improved not only in scale, but also in architecture. This can best be seen by the exponential growth in the number of parameters, for which we present a selection in Fig. 2.5.

→ Fig. 2.5, p. 10

Such models are usually trained in a 2-step procedure, beginning with *pre-training* on high amounts of unlabeled data, and are afterwards *fine-tuned* using a smaller amount of task-specific data. Notable examples include GPT [36], which popularized this paradigm, BERT [37] or T5 [38]. At this scale, language models became *large*. There is no standard definition which sets large language models apart from other language models, but they are widely understood to have a number of parameters at least in the billions (and recently even trillions).

2.4.1. Few-Shot Learning

With the release of GPT-3, LLMs reached a scale were a novel emergent property appeared: *few-shot learning* and relatedly, *prompting* [39, 40, §1.1]. No longer did the models need to be trained on a specific task. Instead, only the task description (*zero-shot*), or additionally one (*one-shot*) or more examples (*few-shot*) can be given as input (*prompt*) to the model. For example, the sentence "*question* in Pig Latin is *estionquay*" would serve as a prior example for the task "What is *synthesis* in Pig Latin?". The term *few-shot learning* is a bit misleading though, since the model itself isn't being adjusted. Instead of learning permanently, the model infers how to solve the task at hand through the given context information in the examples. This also opened the door to use, what essentially was only a text completion engine, for other tasks as well, just by altering the natural language instructions. Note that this doesn't discount fine-tuning, as both techniques can be used alongside each other. Fine-tuning especially remains popular for specialized tasks like code generation or to aid with preventing misuse [41, 38].

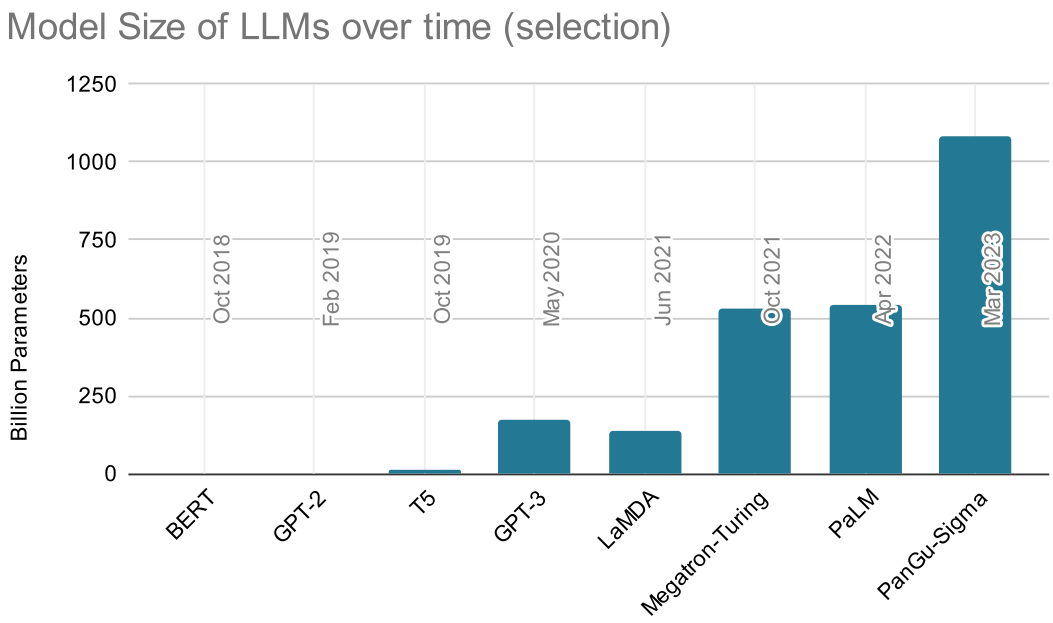


Figure 2.5.: The evolution of model size of LLMs over the past few years (selection).

While this technique quickly gained attention, access to state-of-the-art LLMs like GPT-3 was limited at first. Only after ChatGPT was opened to the public by OpenAI did these capabilities really enter the public conscience and garner a lot of attention, even in other disciplines [42, 43][1].

2.4.2. LLMs for Code Generation

Recently, there have been a lot of promising results in using LLMs for code generation. One of the first successful models specifically tuned for code generation was Codex [44], delivering results much better than its base model (GPT-3) on the benchmarks used. It was originally only trained for Python code generation using public code from GitHub. This was later expanded to other programming languages and commercialized under the name GitHub Copilot. AlphaCode [45] on the other hand, focused on generating code for programming competitions. It was first pre-trained on code from several common programming languages like C++, Java and Python, then fine-tuned on programming competition exercises, which included natural language descriptions as well as solutions. The problems found in programming competitions include hard, algorithmic questions, which presents a challenge not only to human but also AI programmers. The success of

[1]The GPT-3 API was made public in November 2021, but mostly provided text completion as a paid service. The much easier, as well as free-to-use ChatGPT was made public in November 2022.

these models are what motivated us to use LLMs for Verilog code generation, using LTL instead of natural language specifications.

In this work we will be be using publicly available models from the PaLM2 [9] and GPT3.5 [10] model families. Details about the exact model architectures and training data for the specific models we are using are not available in detail, but both are using pre-trained Transformer architectures and are specifically trained for code understanding and generation [46, 47]. We presume that Verilog code will only be a small portion of the training data in comparison to more common programming languages, but at least Google states that PaLM2 can also generate Verilog code [46]. The only model fine-tuned for generating Verilog code known to us is the relatively recent VeriGen model, which has been shown to slightly outperform GPT3.5 and would be interesting to explore in a follow-up [48, 49].

2.5. Related Work

With the rise of Artificial Neural Networks, and more recently LLMs, there has been a surge in attempts to effective apply these advanced machine learning techniques for symbolic reasoning problems.

Austin et al. [50] evaluated LLMs with few-shot learning for use for program synthesis, by specifying test cases (assertions) alongside the natural language description. While not dissimilar to our work, we are focusing on hardware synthesis for use with LTL specifications, which can be formally verified.

Schmitt, Hahn, Rabe, and Finkbeiner [51] directly trained a hierarchical Transformer on specification patterns to generate satisfying circuits in AIGER format. However they limited those patterns to a maximum of five inputs and outputs. In a follow-up they explored repairing partial solutions using a similar approach [52].

Hahn, Schmitt, Kreber, Rabe, and Finkbeiner [53] trained a Transformer model to generate satisfying traces from LTL formulas directly (and not LTL specifications). While it can be used to get examples of correct system behavior, no programs can be constructed from these, as they lack a reactive component.

Efforts have also been made to use machine learning techniques to develop appropriate heuristics for use in LTL synthesis. This would allow exact solvers to more efficiently generate solutions. An early attempt was made by Křetínský, Manta, and Meggendorfer [54] by using Q-learning, a classic technique from reinforcement learning, on the parity game graph. They compared several different features to be used for the reward function. A follow-up paper used various semantic features to train a Support Vector Machine as a classifier to guide the solution constructed. It was also demonstrated how this could be used to improve the LTL synthesis tool Strix [55]. Camacho and McIlraith [56] similarly used deep Q-learning, where the Q-function is approximated by a neural network.

Vasudevan et al. [57] used Graph Neural Networks to learn a semantic representation from Verilog RTL code (a high abstraction level within Verilog) and used this for several

tasks relating to hardware verification, like predicting test coverage and generating additional tests.

Our approach differentiates itself through three key aspects. (1) The use of pre-trained LLMs for LTL synthesis, which hasn't been explored so far. (2) The novel end-to-end approach going directly from LTL specifications to Verilog. (3) Utilizing the characteristic of parameterized specifications, we use solutions for smaller parameters to generate ones for larger parameters.

Chapter 3

Experimental Setup

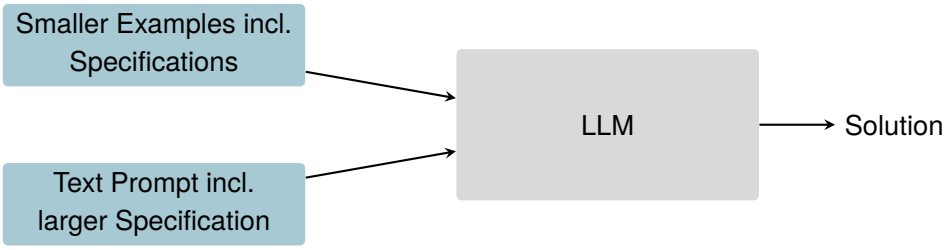


Figure 3.1.: High-level overview of our method

The goal of this work is to investigate whether it is feasible to use LLMs for synthesizing large instances of parametric specifications. *Large* here is referring to the parameter value, not to the size of the raw parameterized specification. The LLM will be shown up to two implementations for smaller parameter values (usually powers of two) and their corresponding specifications. Then, it will be tasked to create a solution for a larger parameter value. A rough overview of this method can be seen in Fig. 3.1. → Fig. 3.1, p. 13

3.1. Detailed Setup

Looking at this process in a bit more detail, several key steps can be identified. The benchmarks must be carefully selected in a way which allows us to obtain reference solution to be used in the prompt later. Afterwards, all potential solutions must be verified for correctness. This process is portrayed in Fig. 3.2. → Fig. 3.2, p. 14

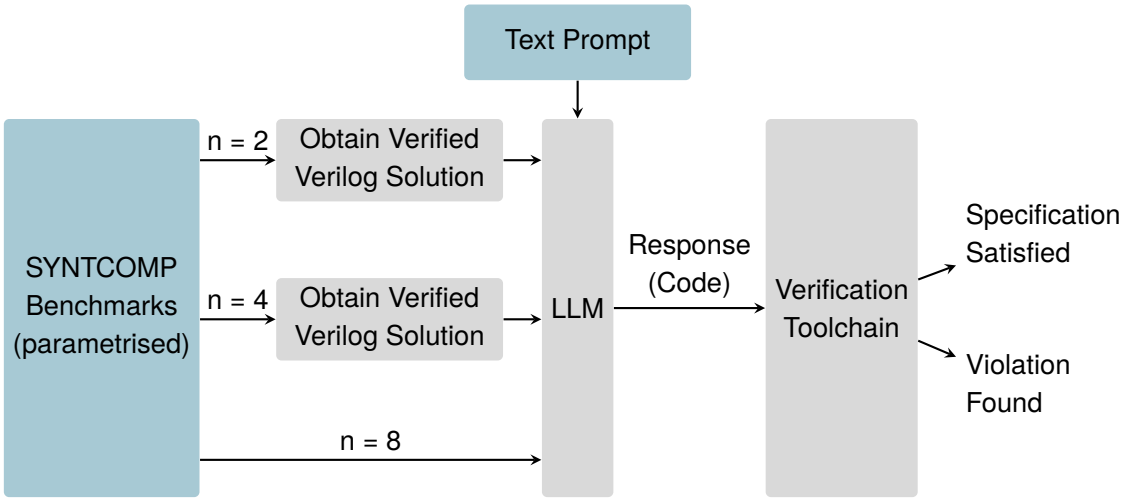


Figure 3.2.: Detailed overview of our method

3.2. Few-Shot Benchmarks

We used the SYNTCOMP benchmarks [23] as a basis for our experiments, deriving three new benchmarks from it. First, a qualitative benchmark with human-written parametric solutions we name *SC-Parametric-Human* (5 specifications) and the two more quantitative benchmarks *SC-Parametric-BoSy* (15 specifications) and *SC-Parametric-Strix* (23 specifications). Every single entry consists of the TLSF specification file as well as two implementation parameter values. For *SC-Parametric-Human*, the examples as well as parameter values were hand-picked, and there exist reference solutions we implemented by hand. The parameter values for *SC-Parametric-Strix* (*SC-Parametric-BoSy*) were picked automatically, such that they are the largest power of two for which Strix (BoSy) can synthesize an implementation in under one minute. We speculate that the restriction to powers of two will further bring out any patterns in the implementation which might appear when scaling the parameter value. The benchmarks were additionally filtered to only contain one parameter as not to introduce additional complexity. The exact parameter values are listed in Appendix B.1.

→ Appendix B.1, p. 43

3.2.1. Obtaining a Verified Verilog Solution

We employ several strategies to obtain the examples which will be included in the prompt. The most straight-forward way is to use classical synthesis tools like Strix [18, 19] or BoSy [14, 15], which will be able to synthesize an implementation for the smaller parameter values used for the examples.

Strix unfortunately doesn't provide a built-in option for generating a Verilog implementation. In order to still be able to use Strix in our experiments we decided to build

a simple translation tool ourselves. This tool is able to take in the AIGER output from Strix and convert it to Verilog code. It does this by directly encoding the AIGER graph. → Fig. 2.3, p. 7 The latches are represented by registers which are updated in synchrony with the clock. The expressions used for updating the state and determining the output values are recursively built from the graph, translating the AND/NOT gates to the corresponding operators in Verilog. Only small optimizations, like double negative elimination are applied. For more details please see the source code provided in Appendix A.2. → Appendix A.2, p. 41

BoSy on the other hand includes a Verilog target option, allowing us to use their implementation directly. It is generated by directly encoding the DFA, which is the internal solution representation, into a Verilog module [58]. To that end, the state space is represented by a multi-bit register (as wide as necessary). The state transitions are a also a direct translation: By discriminating over the state space and inputs in each time step (clock tick), we get the next state (implemented through a series of comparisons). Our experiments have shown that this built-in translation also gave the best results, for more information see Sect. 4.2. → Sec. 4.2, p. 21

When using the *SC-Parametric-Human* dataset, we will also be comparing those tools against our reference solutions, which are easily generalizable, as they are instantiated from the parametric solutions. We expect that these will result in the best performance, as the modules are already written with generalization in mind. We believe those solutions are rather typical for a human programmer without being overly complicated. These implementations can be viewed in Appendix A.1. → Appendix A.1, p. 39

For a direct comparison of the code generated, please see Fig. 3.3. → Fig. 3.3, p. 16

3.3. Prompting Setup

We primarily used two models from the PaLM2 and GPT3.5 model families in our testing. Each of those models provides a way of entering structural input, which we used for each model respectively. Both of those are chat-based models, where each message is a structural unit representing the past conversation/history.

3.3.1. PaLM2

We used the `chat-bison@001`, part of the PaLM2 model family from Google [9]. It's a chat-based model which provides a maximum of 1024 output tokens as well as a similar amount for context and examples (though we were not able to find an exact number). It provides a way to structurally give explicit examples to the model to use as context. The (usually two) examples were always input in that way, separated from the actual task.

```
module detector(r_0, r_1, g);
  input r_0;
  input r_1;
  output g;
  reg [0:0] state;

  assign g = ((state == 1) && r_1 ||
↪  (state == 0) && (!r_1 && r_0)) ? 1 :
↪  0;

  initial
  begin
    state = 0;
  end
  always @($global_clock)
  begin
    case(state)
      0: if (!r_0)
           state = 0;
         else
           state = 1;

      1: if (!(r_1 && !r_0))
           state = 1;
         else
           state = 0;

    endcase
  end
endmodule
```

```
module detector (
  input r_0,
  input r_1,
  output reg g
);
  reg l0;
  reg l1;
  initial begin
    l0 = 0;
    l1 = 0;
  end
  assign g = (!((l1) & (!r_1) & !l0)) &
↪  !l0;
  always @(posedge $global_clock) begin
    l0 <= (!((!l1) & r_0)) & !((l1) &
↪  !((!((l1) & (!r_1) & !l0)) & !l0));
    l1 <= !((!((!l1) & r_0)) & !((l1) &
↪  (!r_1) & !l0));
  end
endmodule
```

```
module detector(
  input [1:0] r,
  input clk,
  output reg g
);
  reg [1:0] state;
  initial state = '0;
  always @(posedge clk) begin
    state = state | r;
    g = 0;
    if(state == '1) begin
      g = 1;
      state = '0;
    end
  end
endmodule
```

Figure 3.3.: A Verilog implementation as synthesized by BoSy (top left) and Strix (top right), as well as our reference solution (bottom)

3.3.2. GPT3.5

From the GPT3.5 model family (OpenAI) we use the `gpt-3.5-turbo-16k` chat model [10, 59]. It provides a much larger context window of 16k tokens which is useful for some of the larger examples. Our prompt consists of the system instruction (additional instructions on how the task should be carried out), the examples (which are passed as prior messages) and the actual task description with the LTL formula.

We provided an example for the GPT3.5 prompt in Fig. 3.5. The prompt used for PaLM2 only differs slightly and is included in Appendix C.1.

→ Fig. 3.5, p. 18
→ Appendix C.1, p. 73

3.4. Verification Workflow

A suite of open-source tools enables us to verify whether the generated solutions are actually correct. The inner workings of this toolchain are described here. We also provide a visual overview in Fig. 3.4.

→ Fig. 3.4, p. 17

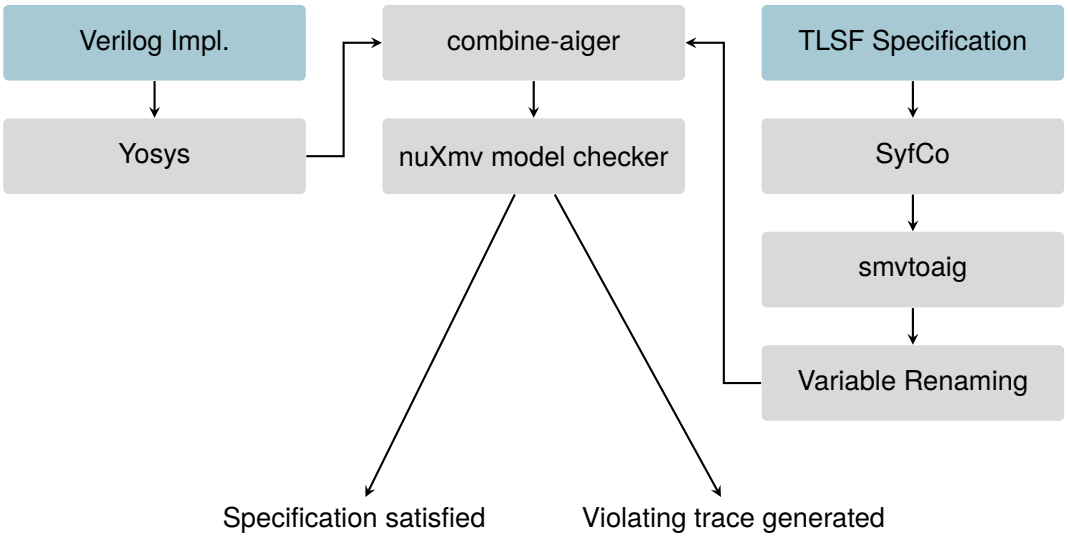


Figure 3.4.: Verification Toolchain

We begin by converting the Verilog code to AIGER using Yosys [29]. Next the TLSF Specification file is converted to standard LTL using SyfCo [13] and then translated into an AIGER monitor file, which encodes the specification in the AIG. We then fix some inconsistencies with the variable naming convention to align it to the one Yosys uses. After that we combine the AIGER files for the implementation and specification into one using `combine-aiger` [60]. This file can then be checked using the nuXmv model checker [28], which will either correctly verify the implementation or output a violating trace ("counterexample"). If nuXmv is not able to verify the solution for correctness in

five minutes, we terminate the process resulting in a timeout. This whole process is encapsulated in a bash file (see Appendix A.2).

→ Appendix A.2, p. 41

SYSTEM You are an expert in writing correct verilog code, which will fulfill certain formal properties specified in LTL. Only reply with the correct verilog module matching the specification and nothing else.

USER Please write a Verilog module for n=2 fulfilling the following specification. Make sure the code is fully synthesizable:
`G (F r_0) && G (F r_1) <-> G (F g)`

ASSISTANT

```
module detector(r_0, r_1, g);
...
endmodule
```

USER Please write a Verilog module for n=4 fulfilling the following specification. Make sure the code is fully synthesizable:
`G (F r_0) && G (F r_1) && G (F r_2) && G (F r_3) <-> G (F g)`

ASSISTANT

```
module detector(r_0, r_1, r_2, r_3, g);
...
endmodule
```

USER Please write a Verilog module for n=8 fulfilling the following specification. Make sure the code is fully synthesizable:
`G (F r_0) && G (F r_1) && G (F r_2) && G (F r_3) && G (F r_4) && G (F r_5) && G (F r_6) && G (F r_7) <-> G (F g)`

Figure 3.5.: The prompt used for GPT3.5.

Chapter 4

Experimental Results

Due to the novel use of LLMs for Reactive Synthesis, this work will be exploring the feasibility and potential of this technique. To that end, we run several experiments to examine different dimensions of that technique. We perform a base experiment and then run several variations that expand on the baseline results. We examine in which cases our method is successful, trying to answer when and why it fails. For these results, note that the problems found in *SC-Parametric-Strix* are in most cases harder than the ones found in *SC-Parametric-BoSy* (larger parameter values), since Strix outperforms BoSy in most instances.

4.1. Baseline Results

In order to establish a baseline, the default prompts were ran on the different datasets, as detailed in Sect. 3.3. For both PaLM2 and GPT3.5, best-of-k runs ($k \in \{1, 3, 5\}$) were performed, meaning k different choices were generated from the LLM, from which the best result was kept. The best result for both *SC-Parametric-Human* and *SC-Parametric-BoSy* was performed by GPT3.5 with $k = 5$ and $k = 3$, achieving scores of 4/5 and 3/15. For *SC-Parametric-Strix* both PaLM2 and GPT3.5 predicted a correct solution for 2/23 benchmarks (for values of $k = 3$ and $k = 5$). In general, these results show GPT3.5 outperforming PaLM2 by a slight margin, which we expected due to the larger context window. Note that the process of generating a response is not deterministic, so a higher value for k stabilizes the results between runs and improves the chance of producing a correct solution. But even with a high k, the overall success rate is still low, which we will improve upon in other experiments.

→ Sec. 3.3, p. 15

Detailed result tables for this section can be found in Appendix B.2.1.

→ Appendix B.2.1, p. 46

LLM	Benchmarks	k = 1	k = 3	k = 5
PaLM2	*SC-Parametric-Human*	3/5	3/5	3/5
	SC-Parametric-BoSy	2/15	2/15	2/15
	SC-Parametric-Strix	1/23	2/23	2/23
GPT3.5	*SC-Parametric-Human*	3/5	3/5	4/5
	SC-Parametric-BoSy	2/15	3/15	2/15
	SC-Parametric-Strix	1/23	2/23	2/23

Table 4.1.: Baseline results, presenting the absolute success rate of both PaLM2 and GPT3.5 on the different benchmarks, using a best-of-k approach.

4.1.1. In which cases are correct solutions produced?

Examining for which benchmarks the LLMs actually generate correct solution, a clear pattern emerges. Unsurprisingly, specifications with simpler solutions work much better. In fact, with the exception of the human-written code, which is much easier to scale, it is exclusively "simple" specifications for which correct code is produced. `mux`, for example, has a relatively short, stateless solution, while the specification for `collector_v3` is not very restrictive, even allowing a constant solution (e.g. see Fig. 4.3). In fact, only for seven distinct benchmarks was any correct code produced in any of the experiments: `amba_decomposed_encode`, `collector_v3`, `detector`, `full_arbiter`, `mux`, `shift`, `simple_arbiter`.

→ Fig. 4.3, p. 22

4.1.2. Why do benchmarks fail?

Benchmarks fail for a variety of reasons, which can largely be grouped into three categories. (1) the prompt exceeding the context window, (2) no syntactically code was produced and (3) correct code was produced, but it violated the specification. (1) will probably improve over time as models expand their context window, but we can also employ techniques to shorten prompts, e.g. by reducing the number of examples. (2) can happen when no code was produced at all, syntactical errors were exhibited, features unsupported by Yosys were used in the code (for example SystemVerilog assertions), or the module definition did not match what is required for verification. We found this to be heavily influenced by the prompt used and that this error can be further reduced by supplying feedback to the model, which would be interesting to explore in the future. For (3), improving these results is a lot more difficult, as often the errors in the code are a lot more nuanced and not trivial to fix. There were also miscellaneous issues such as the model checker timing out, but this only happened in a small number of cases. To be more specific, these groups correspond to the following result codes (as listed in Appendix B.2): (1) corresponds to **AI_ERROR**, (2) to the codes **ERROR_COMBINE**

→ Appendix B.2, p. 46

_AIGER, ERROR_CONVERT_TO_AIGER, NO_CODE, and (3) to **FALSE_RESULT**.

We compiled the errors from our baseline results for $k = 1$ and $k = 5$ ($k = 3$ is excluded for brevity) in Tbl. 4.2. Note that for $k > 1$, the results will be a bit skewed towards error group (3) as this is preferable to error group (2) and only the better result was kept.

→ Tbl. 4.2, p. 21

LLM	Benchmarks	(1)	(2)	(3)	Correct
PaLM2	*SC-Parametric-Human*	0/5	2/5	0/5	3/5
	SC-Parametric-BoSy	4/15	3/15	6/15	2/15
	SC-Parametric-Strix	12/23	9/23	0/23	1/23
GPT3.5	*SC-Parametric-Human*	0/5	2/5	0/5	3/5
	SC-Parametric-BoSy	4/15	2/15	7/15	2/15
	SC-Parametric-Strix	5/23	16/23	0/23	1/23

(a) $k = 1$

LLM	Benchmarks	(1)	(2)	(3)	Correct
PaLM2	*SC-Parametric-Human*	0/5	2/5	0/5	3/5
	SC-Parametric-BoSy	4/15	1/15	8/15	2/15
	SC-Parametric-Strix	12/23	9/23	0/23	1/23
GPT3.5	*SC-Parametric-Human*	0/5	1/5	0/5	4/5
	SC-Parametric-BoSy	5/15	2/15	6/15	2/15
	SC-Parametric-Strix	12/23	7/23	2/23	2/23

(b) $k = 5$

Table 4.2.: Baseline error rates of all tools on their respective datasets, grouped by the failure reason: (1) Exceeded context window, (2) No syntactically correct code produced and (3) Code is syntactically correct, but violates the specification.

4.2. Comparing Different Methods for Generating Verilog Code with BoSy

When implementing the AIGER-to-Verilog translation, the question emerged whether it provided a benefit to use the same translation for BoSy, instead of using the built-in one. For this we compared four methods of generating Verilog Code.

- **Standard**: Uses the standard BoSy Verilog target option, which just translates the finite state machine.

```
module mux (
  input in_0,
  input in_1,
  input in_2,
  input in_3,
  input select_0,
  input select_1,
  output reg out
);
  assign out = !((!((!((!((select_1) &
↪  (in_2))) & (!((in_0) &
↪  (!select_1))))) & (!select_0))) &
↪  (!((select_0) & (!((!((in_1) &
↪  (!select_1))) & (!((in_3) &
↪  (select_1)))))))));
endmodule
```

```
module collector_v3(finished_0,
↪  finished_1, finished_2, finished_3,
↪  all_finished);
  input finished_0;
  input finished_1;
  input finished_2;
  input finished_3;
  output all_finished;

  assign all_finished = 0;
endmodule
```

Figure 4.3.: On the left: A solution for the `mux` benchmark, produced by Strix ($n = 4$)
On the right: A slightly adapted solution for the `collector_v3` benchmark, produced by BoSy ($n = 4$)

- **AIGER** (*aag*): This uses the BoSy AIGER output and converts it into Verilog in the same way as we do for Strix.
- **Optimized AIGER** (*opt_aag*): Similar to the *aag* option, but an optimization pass is performed using *opt* in Yosys before converting to Verilog.
- **Optimized Verilog** (*opt_verilog*): For this option, the BoSy Verilog output will first be translated into AIGER, then an optimization pass will be performed (both using Yosys), before being converted back into Verilog.

LLM	aag	opt_aag	opt_verilog	standard
PaLM2	1/15	1/15	1/15	2/15
GPT3.5	1/15	2/15	1/15	3/15

Table 4.4.: Comparing different Verilog translation methods using BoSy with *SC-Parametric-BoSy* and GPT3.5. Presented here are the absolute success rates on the *SC-Parametric-BoSy* benchmark, evaluated with $k = 5$ (best-of-5)

Apart from the Verilog translation method, the procedure is identical to the one used in Sect. 4.1, with $k = 5$ being used for increased consistency. Looking at the results in Tbl. 4.4, we can see that the standard method outperforms the others by a slight margin (both using GPT3.5 as well as PaLM2). GPT3.5, for example, is able to predict 3/15

→ Sec. 4.1, p. 19
→ Tbl. 4.4, p. 22

solutions correctly. In comparison, *opt_aag*, *aag* and *opt_verilog* only got 2, 1 and 1 out of 15 benchmarks correct. We believe that the way BoSy generates the Verilog code for the finite state machine makes it more predictable than other methods. The downside of this translation is that it is more verbose than other methods and thus produces larger modules (in terms of both character count and token count). This could pose a problem due to the limited context window in LLMs, however this only occurred for large, complex solution which could not be solved in any case regardless (see `amba_decomposed_lock`, for example). For more detailed results, please refer to Appendix B.2.2.

→ Appendix B.2.2, p. 60

Note that we didn't compare the optimized AIGER circuit for Strix, as it already performs circuit optimization by default [18].

4.3. How does the number of examples given influence the results?

For all of our experiments so far, the LLM was always given two reference solutions (*two-shot*). In this experiment, we varied the number of examples given, evaluating both *one-shot* as well as *zero-shot* performance on the benchmarks. The benefit of only using one example is that the prompt length is much shorter, which aids with some problems exceeding the context window. Zero-shot performance is especially interesting, as no information other than the LTL formula will be available to the LLM. The experimental setup is otherwise identical to the one used in Sect. 4.1, with $k = 3$ being fixed for all runs.

→ Sec. 4.1, p. 19

LLM	Benchmarks	Two-shot	One-shot	Zero-shot
PaLM2	*SC-Parametric-Human*	3/5	4/5	0/5
	SC-Parametric-BoSy	2/15	3/15	2/15
	SC-Parametric-Strix	2/23	3/23	1/23
GPT3.5	*SC-Parametric-Human*	3/5	5/5	1/5
	SC-Parametric-BoSy	3/15	4/15	1/15
	SC-Parametric-Strix	2/23	2/23	2/23

Table 4.5.: Comparing how the number of examples given in the prompt affects the performance. Presented here are the absolute success rates on the different benchmarks, evaluated with $k = 3$ (best-of-3)

For all runs, one-shot performance matched or surpassed both zero-shot and two-shot results. This is surprising, as the two-shot prompts contain more information pertaining to the task. The improvements can also not be explained by the cases where the two-shot prompt exceeded the context window size, as those benchmarks could not be solved in

either case. We hypothesize that the substantial improvements over zero-shot are mainly caused by the fact that the reference solutions contains a module definition as well as the name of the module, which in itself contains semantic information. This is further supported by the experiments performed in Sect. 4.5.1. However this does not explain the improvement over two-shot prompting. Our hypothesis is that because the shorter prompt contains less code for the LLM to use for their solution, it is more likely to adapt a solution it has seen in training instead of using the code snippets in the prompt. The latter requiring to actually use the patterns in the code to infer a solution. We further examine the first option (using similar solutions from the training data) in Sect. 4.5.1.

→ Sec. 4.5.1, p. 25
→ Sec. 4.5.1, p. 25

The detailed zero-shot and two-shot data for this section is available in Appendix B.2.1, the one-shot data in Appendix B.2.4.

→ Appendix B.2.1, p. 46
→ Appendix B.2.4, p. 63

4.4. Up to which parameter value are LLMs able to synthesize correct code?

In the few cases where we did see correct results, test were ran in order to probe how far these results could be pushed. For this experiment, all tests were performed with GPT3.5, as the context limit was reached very quickly with PaLM2. The setup starts out identical to the one used for Sect. 4.1 ($k = 5$), trying to generate the solution that was required by the benchmark. Five choices are generated from which the first solution satisfying the specification is picked (if existing). This message containing the solution is then added to the message history / context and a new prompt is built with the parameter value multiplied by two. Afterwards, the same procedure is repeated until the length of the messages surpass the context length. We then remove the oldest messages one at a time, until the prompt fits into the context window again. If, at any point, no correct solution is found, the best parameter value achieved is be returned.

→ Sec. 4.1, p. 19

We found that in the cases were solutions for smaller parameter values were found, we were able to reach solutions for parameters far greater than what is achievable with either Strix or BoSy (up to $n = 512$ from the human reference solutions and $n = 128$ from BoSy/Strix). We expect that this would scale further as well, as the limits reached here can be mostly attributed to the limited context size. Not only did the solutions scale in size, but the expanded specifications did as well.

Benchmark	Self	BoSy	Strix	None
detector	8	0	0	0
full_arbiter	4	0	0	0
mux	128	128	0	128
shift	512	128	128	128
simple_arbiter	16	16	0	0

Table 4.6.: Highest parameter size achieved by GPT3.5 on *SC-Parametric-Human*. Zero signifies that no correct solution was produced. Using the parameter values from *SC-Parametric-Human*, apart from the human-written solutions, we also generated prompts with example solutions from both *BoSy* and *Strix*. *None* refers to a zero-shot prompt.

Another interesting finding is that the example solutions from BoSy performed much better than Strix. For example, for the benchmarks `mux` and `simple_arbiter` the highest parameter values achieved were 128 and 16, while using the example solutions from Strix yielded no correct solutions. This additionally supports the hypothesis that the solutions produced by BoSy are more easily generalizable than the ones generated by Strix. We present the results using the parameter values from the benchmark *SC-Parametric-Human* in Tbl. 4.6. Other results are available in Appendix B.2.3.

→ Tbl. 4.6, p. 25
→ Appendix B.2.3, p. 62

4.5. Examining Semantic Information Contained in the Prompts

4.5.1. To what extent are the example implementations needed for good results?

Prompted by the results presented in Sect. 4.3, we examined the use of a modified zero-shot prompt containing the module definition and a module name corresponding to the problem, but lacking any implementation. The module definition and name are the same as used in the example implementations included in the one- or two-shot prompts. The modified prompt will additionally contain a module definition in the form of `module detector (r_0, r_1, g);`. An full example prompt is available in Appendix C.2.

→ Sec. 4.3, p. 23
→ Appendix C.2, p. 74

Examining the data presented in Tbl. 4.7, it becomes apparent that the modified zero-shot prompt exhibits performance similar to the two-shot results. This result is interesting specifically in relation to the one-shot results, which in most cases surpass our modified zero-shot prompt. The fact that one-shot results still perform better means that while part of the ability of the LLM to generate correct solutions can be explained by semantic information contained in the example, we know that this is not exclusively responsible for all of the positive results.

→ Tbl. 4.7, p. 26

LLM	Benchmarks	Two-shot	One-shot	Zero-shot	Zero-shot (Modified)
PaLM2	*SC-Parametric-Human*	3/5	4/5	0/5	2/5
	SC-Parametric-BoSy	2/15	3/15	2/15	3/15
	SC-Parametric-Strix	2/23	3/23	1/23	2/23
GPT3.5	*SC-Parametric-Human*	3/5	5/5	1/5	2/5
	SC-Parametric-BoSy	3/15	4/15	1/15	3/15
	SC-Parametric-Strix	2/23	2/23	2/23	3/23

Table 4.7.: Comparing the baseline and one-shot results to the modified zero-shot prompt, which includes the module definition. Presented here are the absolute success rates on the different benchmarks, evaluated with $k = 3$ (best-of-3)

4.5.2. Does the LLM exclusively use the information embedded in the LTL atomic proposition naming to solve the problem in the zero-shot setting?

In the zero-shot setting, the prompt only consists of a task description and the LTL formula. But the LTL formula already embeds some semantic meaning within some of the signal names. For example, the multiplexer benchmark has a $\log(n)$-bit `select` signal which selects one of the n inputs to connect to the output. This is also one of the few benchmarks which consistently is solved *zero-shot* and thus from just the formula. Both PaLM2 as well as GPT3.5 are able to correctly generate a Verilog module fulfilling the specification. This begged the question whether this behavior only occurred due to the hints given by the named signals. To verify whether this is the case we renamed the `select` and `input` signals to `a` and `b` respectively. Surprisingly, both PaLM2 and GPT3.5 were still able to correctly solve the problem. PaLM2 additionally was able to identify the specification belonging to a multiplexer, as judged by the module name in the generated solution (see Fig. 4.8 for the PaLM2 Solution, GPT3.5's solution is available in Appendix A.3.1).

→ Fig. 4.8, p. 27
→ Appendix A.3.1, p. 42

We tried to examine publicly available datasets that are known to have been used in training for a lot of LLMs (C4 [61], GitHub), but during a surface-level analysis we were not able to find anything which presented the LTL specification together with similar Verilog code.

```
module mux16(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9,
↪  b_10, b_11, b_12, b_13, b_14, b_15, out);

input a_0, a_1, a_2, a_3;
input b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_10, b_11, b_12, b_13,
↪  b_14, b_15;
output out;
reg out;
always @(*) begin
    case ({a_3, a_2, a_1, a_0})
        4'b0000: out = b_0;
        4'b0001: out = b_1;
        ...
        4'b1111: out = b_15;
    endcase
end
endmodule
```

Figure 4.8.: The solution that was output by PaLM2 for the zero-shot prompt for `mux`. The `select` and `input` signals were renamed to `a` and `b` in the prompt.

4.6. Limitations / Shortcomings

When performing research on LLMs there usually are two ways to obtain access. The first way is to run open-source models on your own, either on dedicated hardware or by using cloud computing providers. Apart from both of those options being expensive or difficult to implement, open-source models often don't reach the state-of-the-art. The second option is to use publicly available APIs, which has a low barrier to entry and provides access to well-performing models. There are several drawbacks to this method though. Models can be opaque and change without notice. This actually happened to us early on with the PaLM2 `codechat-bison@001` model, where there was an unannounced change which caused our results to change significantly (we then switched to our current model, which additionally had a greater context window size). Strict rate-limits can also slow down the experiments, which was especially apparent when working with GPT3.5.

We also wanted to perform more experiments with the GPT-4 model family, which is considered state of the art at the moment [62]. Unfortunately, we were not granted access in time to incorporate these results thoroughly into this work. Early experiments suggest slightly improved performance, especially with the zero-shot prompt. Some result tables for GPT-4 are available in Appendix B.2.6.

→ Appendix B.2.6, p. 69

Chapter 5

Conclusion

In this work, we presented a new approach for synthesizing digital circuits from LTL specifications. We used Large Language Models to directly generate high-level solutions in the Verilog HDL. To this end, we used *few-shot learning*, presenting example solutions for a lower parameter value first, before prompting the model to generate solutions for the same specification, but with an increased parameter value. Our main goal was to examine the feasibility and potential of this approach, exploring different variations on the concept.

We developed an environment to test different prompts on the two LLMs PaLM2 and GPT3.5. This includes a full verification toolchain, which allows us to formally verify the generated solutions for correctness. In the case that a solution cannot be verified, it is able to give feedback on which step in the process caused the error. We provide the full code and datasets in an accompanying git repository (Appendix A.2).

→ Appendix A.2, p. 41

We derived three new benchmarks from the SYNTCOMP [23] benchmarks to use for our approach. *SC-Parametric-BoSy* and *SC-Parametric-Strix* are both based on the abilities of the LTL synthesis tools BoSy and Strix respectively. It contains examples for parameter values up to the highest power of two for which the corresponding tool is able to synthesize a solution. The LLM is later prompted to generate a solution for the next highest power of two. *SC-Parametric-Human* uses reference Verilog implementations we wrote to investigate whether it could be useful to manually write solutions for smaller parameter values and then use an LLM to generate solutions for larger values.

Using different prompts, we showed that this approach is able to surpass what a traditional LTL synthesis tool like Strix is able to achieve (for a limited number of specifications). In these successful cases, we were often able to increase the parameter value by several orders of magnitude while still getting correct results, up to $n = 512$ (in comparison, the highest power of two the state-of-the-art synthesis tool Strix was able to achieve is $n = 8$). This was limited by several factors, including the fact that at these large parameter values, the expanded LTL formulas can get too large, taking up a large

portion of or even exceeding the context window. We touch on possible solutions in the outlook (Sect. 5.1).

→ Sec. 5.1, p. 30

We showed that the best results can be achieved by using a one-shot prompt (containing one example implementation), which exhibited improved performance over both zero-shot and two-shot prompts.

We demonstrated that the *SC-Parametric-Human* benchmarks, in contrast to *SC-Parametric-Strix* and *SC-Parametric-BoSy*, consistently exhibited good performance, producing a correct solution for 5/5 benchmarks in the best case. The best cases for *SC-Parametric-BoSy* and *SC-Parametric-Strix* were 4/15 and 3/23 respectively. This is promising, as it suggests that it might be worthwhile for a human to write a solution for a small parameter value and then using an LLM to generalize to larger parameter values.

However it should be noted that this approach can not directly be compared to these classical tools. The fact that the models used are opaque in their architecture and computational requirements makes a comparison difficult, especially since a solution is not explicitly computed. For the same reason, generating a correct implementation is not directly related to the computational expenditure. The production of a correct implementation cannot be guaranteed, making it unsuitable for use on its own. A hybrid approach however, e.g. by using it alongside other solvers or integrating it into a programming environment, might prove useful (Sect. 5.1).

→ Sec. 5.1, p. 30

The question of why this approach only worked for some specifications and not others could not be exhaustively answered, but we believe that it is the result of both the complexity of the specification, as well as the frequency in the training data of the LLMs. As explainability for LLMs is still an open research question, we hope that future work will bring further light into the situation.

5.1. Outlook

We identified several ways to further build on our approach, which we will present here.

Explore LLMs fine-tuned on LTL/Verilog. In this work, we have only been using general-purpose LLMs like PaLM2 and GPT3.5, which have not been specifically fine-tuned to perform well with Verilog code. Using a fine-tuned model like VeriGen [48, 49] or fine-tuning a model ourselves might boost performance, reducing especially syntactical errors.

Using the LLM in an integrated environment such as Copilot. The benchmarks which performed especially well were common circuits (e.g. a multiplexer, bit-shift etc.) which leads us to believe that a lot of training data for these circuits was available. If there was more training data available for other common cases this could become a useful tool for practical applications where the LLM could work hand-in-hand with a human programmer. As all solutions are formally verified, this cannot produce unreliable code.

This also ties in to the proposition of using an LLM fine-tuned on Verilog Code and/or LTL specifications.

Exploring different specification languages. In this work, the prompts only included LTL formulas in its basic form. With higher parameter values, LTL can get very verbose and doesn't leverage the fact that we are only using parameterized specifications. TLSF in comparison is not only less verbose, but also allows to specify complicated properties in a compact way. For example, it is common to require *mutual exclusion* (only one signal is allowed to be high at the same time). If this was explicitly stated, like is possible in TLSF, this is information which the LLM could leverage. However, even less training data exists for TLSF and it is possible that the format might be too complex with its various sections (given the sparsity of relevant data, the difference between e.g. `REQUIRE`, `PRESET` and `ASSUME` might become difficult to distinguish). Compromising between those extremes, e.g. by limiting TLSF, might become necessary.

Repairing partial solutions. As the LLMs often produce some sensible results, even if not completely correct, it might be possible to repair the partial solutions. There has been some existing work in this regard, like the framework CirFix [63], which already operates on Verilog code. Similarly, Cosler, Schmitt, Hahn, and Finkbeiner [52] focused on repairing circuits directly with regard to the formal specification. Using these tools on our wrong solutions might improve performance in the cases where syntactically correct code was produced but which violated the specification.

Bibliography

[1] Amir Pnueli. "The Temporal Logic of Programs". In: *FOCS 1977*. IEEE Computer Society, 1977, pp. 46–57. URL: https://doi.org/10.1109/SFCS.1977.32.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 9780262026499.

[3] Alonzo Church. "Applications of recursive arithmetic to the problem of circuit synthesis Alonzo Church". In: *Summaries of talks presented at the Summer Institute for Symbolic Logic* 1.2 (1957), pp. 3–50.

[4] A. Pnueli and R. Rosner. "On the Synthesis of a Reactive Module". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 179–190. ISBN: 0897912942. URL: https://doi.org/10.1145/75277.75293.

[5] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) Designs". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–380. ISBN: 978-3-540-31622-0.

[6] Jordan Hoffmann et al. "Training Compute-Optimal Large Language Models". In: *CoRR* abs/2203.15556 (2022). URL: https://doi.org/10.48550/arXiv.2203.15556.

[7] Jack W. Rae et al. "Scaling Language Models: Methods, Analysis & Insights from Training Gopher". In: *CoRR* abs/2112.11446 (2021). URL: https://arxiv.org/abs/2112.11446.

[8] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *CoRR* abs/2204.02311 (2022). URL: https://doi.org/10.48550/arXiv.2204.02311.

[9] Rohan Anil et al. "PaLM 2 Technical Report". In: *CoRR* abs/2305.10403 (2023). URL: https://doi.org/10.48550/arXiv.2305.10403.

[10] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). URL: https://arxiv.org/abs/2005.14165.

[11] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[12] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. 07/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, Oct. 2007.

[13] Swen Jacobs, Felix Klein, and Sebastian Schirmer. "A High-Level LTL Synthesis Format: TLSF v1.1". In: *Electronic Proceedings in Theoretical Computer Science* 229 (Nov. 2016), pp. 112–132. URL: https://doi.org/10.48550/arXiv.1604.02284.

[14] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. "Encodings of Bounded Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 354–370. URL: https://doi.org/10.1007/978-3-662-54577-5%5C_20.

[15] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. "BoSy: An Experimentation Framework for Bounded Synthesis". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 325–332. URL: https://doi.org/10.1007/978-3-319-63390-9%5C_17.

[16] Mikoláš Janota. "RAReQS: Recursive Abstraction Refinement QBF Solver". In: Lisbon, Portugal: INESC-ID, Apr. 2012.

[17] Sven Schewe and Bernd Finkbeiner. "Bounded Synthesis". In: *Automated Technology for Verification and Analysis*. Ed. by Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 474–488. ISBN: 978-3-540-75596-8.

[18] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. "Strix: Explicit Reactive Synthesis Strikes Back!" In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 578–586. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_31.

[19] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. "Practical synthesis of reactive systems from LTL specifications via parity games". In: *Acta Informatica* 57.1-2 (2020), pp. 3–36. URL: https://doi.org/10.1007/s00236-019-00349-3.

[20] J. Richard Büchi and Lawrence H. Landweber. "Solving sequential conditions by finite-state strategies". In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311.

[21] *SYNTCOMP 2023 Results*. 2023. URL: http://www.syntcomp.org/syntcomp-2023-results/ (visited on 07/21/2023).

[22] *SYNTCOMP 2022 Results*. 2022. URL: http://www.syntcomp.org/syntcomp-2022-results/ (visited on 07/21/2023).

[23] Swen Jacobs et al. "The Reactive Synthesis Competition (SYNTCOMP): 2018-2021". In: *CoRR* abs/2206.00251 (2022). URL: https://doi.org/10.48550/arXiv.2206.00251.

[24] Andreas Kuehlmann and Florian Krohm. "Equivalence Checking Using Cuts and Heaps". In: *Proceedings of the 34th Annual Design Automation Conference*. DAC '97. Anaheim, California, USA: Association for Computing Machinery, 1997, pp. 263–268. ISBN: 0897919203. URL: https://doi.org/10.1145/266021.266090.

[25] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40. ISBN: 978-3-642-14295-6.

[26] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. "Robust Boolean reasoning for equivalence checking and functional property verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.12 (2002), pp. 1377–1394. DOI: 10.1109/TCAD.2002.804386.

[27] Swen Jacobs. "Extended AIGER Format for Synthesis". In: *CoRR* abs/1405.5793 (2014). URL: http://arxiv.org/abs/1405.5793.

[28] Roberto Cavada et al. "The nuXmv Symbolic Model Checker". In: *CAV*. 2014, pp. 334–342.

[29] Clifford Wolf, Johann Glaser, and Johannes Kepler. *Yosys - A Free Verilog Synthesis Suite*. 2013. URL: https://yosyshq.net/yosys/about.html.

[30] "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language". In: *IEEE Std 1800-2005* (2005), pp. 1–648. DOI: 10.1109/IEEESTD.2005.97972.

[31] Fred Jelinek and Robert L. Mercer. "Interpolated estimation of Markov source parameters from sparse data". In: *Proceedings of Workshop Pattern Recognition in Practice*. Ed. by Edzard S. Gelsema and Laveen N. Kanal. Amsterdam, The Netherlands: North Holland, 1980.

[32] Dan Jurafsky and James H. Martin. *Speech and Language Processing*. 3rd ed. draft. 2023. URL: https://web.stanford.edu/~jurafsky/slp3/ (visited on 09/13/2023).

[33] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[34] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). URL: http://arxiv.org/abs/1706.03762.

[35] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1310–1318. URL: https://proceedings.mlr.press/v28/pascanu13.html.

[36] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. *Improving language understanding by generative pre-training*. 2018.

[37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). URL: http://arxiv.org/abs/1810.04805.

[38] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *CoRR* abs/1910.10683 (2019). URL: http://arxiv.org/abs/1910.10683.

[39] Jason Wei et al. "Emergent Abilities of Large Language Models". In: *CoRR* abs/2206.07682 (2022). URL: https://doi.org/10.48550/arXiv.2206.07682.

[40] Rishi Bommasani et al. "On the Opportunities and Risks of Foundation Models". In: *CoRR* abs/2108.07258 (2021). URL: https://arxiv.org/abs/2108.07258.

[41] Hugo Touvron et al. "Llama 2: Open Foundation and Fine-Tuned Chat Models". In: *CoRR* abs/2307.09288 (2023). URL: https://doi.org/10.48550/arXiv.2307.09288.

[42] OpenAI Blog. *Introducing ChatGPT*. Nov. 2022. URL: https://openai.com/blog/api-no-waitlist (visited on 08/15/2023).

[43] OpenAI Blog. *OpenAI's API now available with no waitlist*. Nov. 2021. URL: https://openai.com/blog/api-no-waitlist (visited on 08/15/2023).

[44] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). URL: https://arxiv.org/abs/2107.03374.

[45] Yujia Li et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (Dec. 2022), pp. 1092–1097. URL: https://doi.org/10.1126%2Fscience.abq1158.

[46] Google Blog. *Introducing PaLM 2*. URL: https://blog.google/technology/ai/google-palm-2-ai-large-language-model/ (visited on 08/25/2023).

[47] OpenAI Documentation. *Models*. URL: https://platform.openai.com/docs/models (visited on 08/25/2023).

[48] Shailja Thakur et al. "Benchmarking Large Language Models for Automated Verilog RTL Code Generation". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*. IEEE, 2023, pp. 1–6. URL: https://doi.org/10.23919/DATE56975.2023.10137086.

[49] Shailja Thakur et al. "VeriGen: A Large Language Model for Verilog Code Generation". In: *CoRR* abs/2308.00708 (2023). URL: https://doi.org/10.48550/arXiv.2308.00708.

[50] Jacob Austin et al. "Program Synthesis with Large Language Models". In: *CoRR* abs/2108.07732 (2021). URL: https://arxiv.org/abs/2108.07732.

[51] Frederik Schmitt, Christopher Hahn, Markus N. Rabe, and Bernd Finkbeiner. "Neural Circuit Synthesis from Specification Patterns". In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan. 2021, pp. 15408–15420. URL: https://proceedings.neurips.cc/paper/2021/hash/8230bea7d54bcdf99cdfe85cb07313d5-Abstract.html.

[52] Matthias Cosler, Frederik Schmitt, Christopher Hahn, and Bernd Finkbeiner. "Iterative Circuit Repair Against Formal Specifications". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=SEcSahl0Ql.

[53] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. "Teaching Temporal Logics to Neural Networks". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=dOcQK-f4byz.

[54] Jan Křetínský, Alexander Manta, and Tobias Meggendorfer. "Semantic Labelling and Learning for Parity Game Solving in LTL Synthesis". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Vol. 11781. Lecture Notes in Computer Science. Springer, 2019, pp. 404–422. URL: https://doi.org/10.1007/978-3-030-31784-3%5C_24.

[55] Jan Křetínský, Tobias Meggendorfer, Maximilian Prokop, and Sabine Rieder. "Guessing Winning Policies in LTL Synthesis by Semantic Learning". In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Cham: Springer Nature Switzerland, 2023, pp. 390–414. ISBN: 978-3-031-37706-8.

[56] Alberto Camacho and Sheila A. McIlraith. "Towards Neural-Guided Program Synthesis for Linear Temporal Logic Specifications". In: *CoRR* abs/1912.13430 (2019). URL: http://arxiv.org/abs/1912.13430.

[57] Shobha Vasudevan et al. "Learning Semantic Representations to Verify Hardware Designs". In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan. 2021, pp. 23491–23504. URL: https://proceedings.neurips.cc/paper/2021/hash/c5aa65949d20f6b20e1a922c13d974e7-Abstract.html.

[58] Reactive Systems Group. *BoSy Source Code for Verilog Target*. 2021. URL: https://github.com/reactive-systems/bosy/blob/a1bb68d4232c5e58d78793706d8d08c556888bad/Sources/TransitionSystem/ExplicitState.swift (visited on 07/30/2023).

[59] OpenAI Blog. *Function calling and other API updates*. June 2023. URL: https://openai.com/blog/function-calling-and-other-api-updates (visited on 08/15/2023).

[60] Leander Tentrup. *combine-aiger*. URL: https://github.com/SYNTCOMP/combine-aiger (visited on 07/30/2023).

[61] Allen Institute for AI. URL: https://c4-search.apps.allenai.org (visited on 08/15/2023).

[62] OpenAI. "GPT-4 Technical Report". In: *CoRR* abs/2303.08774 (2023). URL: https://doi.org/10.48550/arXiv.2303.08774.

[63] Priscila Santiesteban, Yu Huang, Westley Weimer, and Hammad Ahmad. "CirFix: Automated Hardware Repair and its Real-World Applications". In: *IEEE Transactions on Software Engineering* 49.7 (2023), pp. 3736–3752. DOI: 10.1109/TSE.2023.3269899.

Appendix A

Verilog Solutions and Source Code Files

A.1. Parametric Verilog Solutions

A.1.1. Shift (shift.tlsf)

```
module shift #(parameter n = 2) (
  input [n-1:0] in,
  input clk,
  output [n-1:0] out
);
  always @(*) begin
    out = {in[n-2:0], in[n-1]};
  end
endmodule
```

A.1.2. Multiplexer (mux.tlsf)

```
module mux #(parameter n = 4) (
  input [n-1:0] in,
  input [$clog2(n)-1:0] select,
  input clk,
  output reg out
);
  wire [0:$clog2(n)-1] reversed = select;
  assign out = in[reversed];
endmodule
```

A.1.3. Detector (detector.tlsf)

```
module detector #(parameter n = 2) (
  input [n-1:0] r,
  input clk,
  output reg g
);
  reg [n-1:0] state;
  initial state = '0;
  always @(posedge clk) begin
    state = state | r;
    g = 0;
    if(state == '1) begin
      g = 1;
      state = '0;
    end
  end
endmodule
```

A.1.4. Simple Arbiter (simple_arbiter.tlsf)

```
module simple_arbiter #(parameter n = 4) (
  input [n-1:0] r,
  input clk,
  output reg [n-1:0] g
);
  initial g = 1;
  always @(posedge clk) begin
    g = {g[n-2:0], g[n-1]};
  end
endmodule
```

A.1.5. Full Arbiter (full_arbiter.tlsf)

Please note that the specification for the full arbiter is actually a restriction on the specification of the simple arbiter. This means that this solution would also be a valid solution for the simple arbiter.

```
module full_arbiter #(parameter n = 4) (
  input [n-1:0] r,
  input clk,
  output reg [n-1:0] g
);
//Needed because of lack of break; statement support
  reg brk = 0;
  reg [n-1:0] requests = '0;
  reg [n-1:0] i = '0;
  reg [n-1:0] istop = 0;
  initial g = '0;
  always @(posedge clk) begin
    g = '0;
    for (i = 0; i < n; i = i + 1) begin
      if(!brk || istop - 1 != i)
        requests[i] = requests[i] | r[i];
    end
    if(istop >= n)
      istop = '0;
    brk = 0;
    for (i = 0; i < n; i = i + 1) begin
      //continue where you left off
      if(i >= istop &&
         //loop "break"
         !brk &&
         //only act if there is a request
         requests[i] == 1) begin

        requests[i] = 0;
        g[i] = 1;
        istop = i + 1;
        brk = 1;
      end
    end
    if(brk == 0)
      istop = '0;
  end
endmodule
```

A.2. Source Code Repo

All of the used source code can be found in the accompanying git repository (`https://github.com/Pflyg/ltl-synth-llm`). This most notably includes

- `aigertoverilog.py`, which is responsible for converting the Aiger output to Verilog code (used primarily with Strix)
- `verify.sh`, the bash script verifying the verilog code against the specification. We usually interact with it using the python wrapper script (`verify.py`)

A.3. Experiments

A.3.1. Mux Zero-Shot Solution With Renamed Variables

```
module VerilogModule (
  input wire a_3,
  input wire a_2,
  input wire a_1,
  input wire a_0,
  output wire out,
  input wire [15:0] b
);

  assign out = ( ( (!a_3) && (!a_2) && (!a_1) && (!a_0) && b[0] ) ||
                ( (!a_3) && (!a_2) && (!a_1) && a_0 && b[1] ) ||
                ( (!a_3) && (!a_2) && a_1 && (!a_0) && b[2] ) ||
                ( (!a_3) && (!a_2) && a_1 && a_0 && b[3] ) ||
                ( (!a_3) && a_2 && (!a_1) && (!a_0) && b[4] ) ||
                ( (!a_3) && a_2 && (!a_1) && a_0 && b[5] ) ||
                ( (!a_3) && a_2 && a_1 && (!a_0) && b[6] ) ||
                ( (!a_3) && a_2 && a_1 && a_0 && b[7] ) ||
                ( a_3 && (!a_2) && (!a_1) && (!a_0) && b[8] ) ||
                ( a_3 && (!a_2) && (!a_1) && a_0 && b[9] ) ||
                ( a_3 && (!a_2) && a_1 && (!a_0) && b[10] ) ||
                ( a_3 && (!a_2) && a_1 && a_0 && b[11] ) ||
                ( a_3 && a_2 && (!a_1) && (!a_0) && b[12] ) ||
                ( a_3 && a_2 && (!a_1) && a_0 && b[13] ) ||
                ( a_3 && a_2 && a_1 && (!a_0) && b[14] ) ||
                ( a_3 && a_2 && a_1 && a_0 && b[15] ));

endmodule
```

Appendix B

Result Tables

B.1. Benchmark Parameter Values

The following tables specify the exact parameter values used for *SC-Parametric-Human*, *SC-Parametric-BoSy* and *SC-Parametric-Strix* respectively.

Benchmark	Example Parameter Value 1	Example Parameter Value 2	Parameter Value to generate
detector	2	4	8
full_arbiter	2	3	4
mux	4	8	16
shift	4	8	16
simple_arbiter	2	4	8

Table B.1.: Parameter values used for *SC-Parametric-Human*

Benchmark	Example Parameter Value 1	Example Parameter Value 2	Parameter Value to generate
amba_ decomposed_ arbiter	2	4	8
amba_ decomposed_ encode	4	8	16
amba_ decomposed_ lock	4	8	16
collector_v2	2	4	8
collector_v3	2	4	8
detector	2	4	8
load_balancer	2	4	8
ltl2dba_C2	2	4	8
ltl2dba_alpha	2	4	8
mux	4	8	16
narylatch	2	4	8
prioritized_ arbiter	2	4	8
shift	4	8	16
simple_arbiter	2	4	8
simple_ arbiter_enc	2	4	8

Table B.2.: Parameter values used for *SC-Parametric-BoSy*

Benchmark	Example Parameter Value 1	Example Parameter Value 2	Parameter Value to generate
amba_ decomposed_ arbiter	2	4	8
amba_ decomposed_ encode	4	8	16
amba_ decomposed_ lock	4	8	16
collector_v1	4	8	16
collector_v2	4	8	16
collector_v3	4	8	16
detector	16	32	64
full_arbiter	2	4	8
load_balancer	2	4	8
ltl2dba_C2	16	32	64
ltl2dba_E	4	8	16
ltl2dba_Q	2	4	8
ltl2dba_U1	4	8	16
ltl2dba_alpha	8	16	32
ltl2dba_beta	4	8	16
mux	8	16	32
narylatch	4	8	16
prioritized_ arbiter	4	8	16
prioritized_ arbiter_enc	2	4	8
round_robin_ arbiter	2	4	8
shift	4	8	16
simple_arbiter	8	16	32
simple_ arbiter_enc	2	4	8

Table B.3.: Parameter values used for *SC-Parametric-Strix*

B.2. Result Tables

The following result tables contain several different results codes. This is a short explanation of how they should be interpreted:

- **SUCCESS**: A correct solution was identified.
- **FALSE_RESULT**: The module matches the required definition (ports) but violated the specification.
- **NO_CODE**: No verilog module could be identified in the output from the LLM.
- **TIMEOUT**: The synthesis tool used wasn't able to find solutions for the required examples in time (this is primarily the case when running the *SC-Parametric-Strix* dataset together with BoSy, as Strix is usually much faster at identifying solutions).
- **AI_ERROR**: Unexpected errors occurred with the LLM used. This usually means that the context length was exceeded.
- **AI_RATELIMIT**: We weren't able to get results from the LLM due to rate limiting, even after several retries.
- **ERROR_CONVERT_TO_AIGER**: Yosys wasn't able to convert the Verilog code to Aiger. This usually happens due to syntax errors or unsupported constructs being used.
- **ERROR_COMBINE_AIGER**: The Aiger implementation and monitor files couldn't be combined. In our testing this only happened if the module definition was incorrect, e.g. the ports used had incorrect names / were omitted etc.
- **VERIFICATION_TIMEOUT**: A solution which matched the required definition was identified, but the model checker timed out.

B.2.1. Baseline Results

These are the exact results for the baseline experiments. For each dataset we ran all of the tools (where possible) on the same dataset, to be able to compare results later.

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	ERROR_CONVERT_TO_AIGER	ERROR_COMBINE_AIGER
full_arbiter	ERROR_COMBINE_AIGER	NO_CODE	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
mux	ERROR_COMBINE_AIGER	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER	FALSE_RESULT
shift	SUCCESS	SUCCESS	SUCCESS	ERROR_COMBINE_AIGER
simple_arbiter	SUCCESS	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT_TO_AIGER

Table B.4.: Results for *SC-Parametric-Human* using PaLM2 (single run)

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	FALSE_RESULT	ERROR_COMBINE_AIGER
full_arbiter	ERROR_COMBINE_AIGER	FALSE_RESULT	ERROR_CONVERT_TO_AIGER	ERROR_COMBINE_AIGER
mux	ERROR_COMBINE_AIGER	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER	FALSE_RESULT
shift	SUCCESS	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	SUCCESS	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT

Table B.5.: Results for *SC-Parametric-Human* using PaLM2 (best-of-3)

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
full_arbiter	ERROR_COMBINE_AIGER	FALSE_RESULT	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
mux	ERROR_COMBINE_AIGER	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER	SUCCESS
shift	SUCCESS	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	SUCCESS	SUCCESS	FALSE_RESULT	FALSE_RESULT

Table B.6.: Results for *SC-Parametric-Human* using PaLM2 (best-of-5)

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
full_arbiter	ERROR_COMBINE_AIGER	FALSE_RESULT	NO_CODE	FALSE_RESULT
mux	ERROR_COMBINE_AIGER	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER	FALSE_RESULT
shift	SUCCESS	SUCCESS	SUCCESS	ERROR_CONVERT_TO_AIGER
simple_arbiter	SUCCESS	SUCCESS	ERROR_CONVERT_TO_AIGER	NO_CODE

Table B.7.: Results for *SC-Parametric-Human* using GPT3.5 (single run)

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	FALSE_RESULT	ERROR_COMBINE_AIGER
full_arbiter	ERROR_COMBINE_AIGER	FALSE_RESULT	NO_CODE	FALSE_RESULT
mux	ERROR_COMBINE_AIGER	ERROR_COMBINE_AIGER	ERROR_CONVERT_TO_AIGER	SUCCESS
shift	SUCCESS	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	SUCCESS	SUCCESS	SUCCESS	ERROR_CONVERT_TO_AIGER

Table B.8.: Results for *SC-Parametric-Human* using GPT3.5 (best-of-3)

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
full_arbiter	ERROR_COMBINE_AIGER	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
mux	SUCCESS	ERROR_CONVERT_TO_AIGER	FALSE_RESULT	SUCCESS
shift	SUCCESS	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	SUCCESS	SUCCESS	ERROR_CONVERT_TO_AIGER

Table B.9.: Results for *SC-Parametric-Human* using GPT3.5 (best-of-5)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	NO_CODE
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	ERROR_COMBINE_AIGER
amba_ decomposed_ lock	AI_ERROR	NO_CODE	FALSE_RESULT
collector_v2	FALSE_RESULT	NO_CODE	ERROR_COMBINE_AIGER
collector_v3	SUCCESS	SUCCESS	VERIFICATION_TIMEOUT
detector	FALSE_RESULT	NO_CODE	SUCCESS
load_balancer	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_C2	NO_CODE	NO_CODE	FALSE_RESULT
ltl2dba_alpha	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
mux	NO_CODE	NO_CODE	ERROR_COMBINE_AIGER
narylatch	AI_ERROR	NO_CODE	FALSE_RESULT
prioritized_ arbiter	FALSE_RESULT	NO_CODE	FALSE_RESULT
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	FALSE_RESULT	FALSE_RESULT	ERROR_COMBINE_AIGER
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	ERROR_CONVERT _TO_AIGER

Table B.10.: Results for *SC-Parametric-BoSy* using PaLM2 (single run)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	NO_CODE
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ lock	AI_ERROR	NO_CODE	FALSE_RESULT
collector_v2	FALSE_RESULT	NO_CODE	ERROR_COMBINE_AIGER
collector_v3	SUCCESS	SUCCESS	VERIFICATION_TIMEOUT
detector	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	SUCCESS
load_balancer	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_C2	NO_CODE	NO_CODE	FALSE_RESULT
ltl2dba_alpha	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
mux	NO_CODE	SUCCESS	ERROR_COMBINE_AIGER
narylatch	AI_ERROR	NO_CODE	FALSE_RESULT
prioritized_ arbiter	FALSE_RESULT	NO_CODE	FALSE_RESULT
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	FALSE_RESULT	FALSE_RESULT	ERROR_COMBINE_AIGER
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	ERROR_CONVERT _TO_AIGER

Table B.11.: Results for *SC-Parametric-BoSy* using PaLM2 (best-of-3)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	SUCCESS
amba_ decomposed_ lock	AI_ERROR	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
collector_v2	FALSE_RESULT	NO_CODE	FALSE_RESULT
collector_v3	SUCCESS	SUCCESS	ERROR_CONVERT _TO_AIGER
detector	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
load_balancer	NO_CODE	AI_ERROR	ERROR_CONVERT _TO_AIGER
ltl2dba_C2	FALSE_RESULT	FALSE_RESULT	ERROR_COMBINE_AIGER
ltl2dba_alpha	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
mux	FALSE_RESULT	FALSE_RESULT	SUCCESS
narylatch	AI_ERROR	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
prioritized_ arbiter	FALSE_RESULT	NO_CODE	FALSE_RESULT
shift	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	ERROR_CONVERT _TO_AIGER

Table B.12.: Results for *SC-Parametric-BoSy* using PaLM2 (best-of-5)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	NO_CODE	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	NO_CODE	NO_CODE
amba_ decomposed_ lock	AI_ERROR	NO_CODE	ERROR_CONVERT _TO_AIGER
collector_v2	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
collector_v3	SUCCESS	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
detector	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	NO_CODE
load_balancer	ERROR_CONVERT _TO_AIGER	NO_CODE	NO_CODE
ltl2dba_C2	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	NO_CODE
ltl2dba_alpha	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	NO_CODE
mux	ERROR_CONVERT _TO_AIGER	FALSE_RESULT	SUCCESS
narylatch	AI_ERROR	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
prioritized_ arbiter	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
shift	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	FALSE_RESULT	FALSE_RESULT	NO_CODE
simple_arbiter_ enc	FALSE_RESULT	NO_CODE	ERROR_CONVERT _TO_AIGER

Table B.13.: Results for *SC-Parametric-BoSy* using GPT3.5 (single run)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	NO_CODE	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
amba_ decomposed_ lock	AI_ERROR	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
collector_v2	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
collector_v3	SUCCESS	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
detector	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
load_balancer	ERROR_CONVERT _TO_AIGER	NO_CODE	ERROR_COMBINE_AIGER
ltl2dba_C2	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
ltl2dba_alpha	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
mux	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
narylatch	AI_ERROR	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
prioritized_ arbiter	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
simple_arbiter_ enc	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER

Table B.14.: Results for *SC-Parametric-BoSy* using GPT3.5 (best-of-3)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	FALSE_RESULT
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ lock	AI_ERROR	FALSE_RESULT	ERROR_COMBINE_AIGER
collector_v2	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
collector_v3	SUCCESS	SUCCESS	ERROR_COMBINE_AIGER
detector	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
load_balancer	NO_CODE	AI_ERROR	ERROR_CONVERT _TO_AIGER
ltl2dba_C2	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
ltl2dba_alpha	AI_ERROR	FALSE_RESULT	ERROR_COMBINE_AIGER
mux	ERROR_CONVERT _TO_AIGER	FALSE_RESULT	SUCCESS
narylatch	AI_ERROR	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
prioritized_ arbiter	FALSE_RESULT	NO_CODE	SUCCESS
shift	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	ERROR_CONVERT _TO_AIGER

Table B.15.: Results for *SC-Parametric-BoSy* using GPT3.5 (best-of-5)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	NO_CODE
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	ERROR_COMBINE_AIGER
amba_ decomposed_ lock	AI_ERROR	NO_CODE	FALSE_RESULT
collector_v1	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
collector_v2	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
collector_v3	TIMEOUT	ERROR_CONVERT _TO_AIGER	VERIFICATION_TIMEOUT
detector	TIMEOUT	NO_CODE	VERIFICATION_TIMEOUT
full_arbiter	NO_CODE	AI_ERROR	NO_CODE
load_balancer	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_C2	TIMEOUT	NO_CODE	NO_CODE
ltl2dba_E	TIMEOUT	VERIFICATION_TIMEOUT	ERROR_COMBINE_AIGER
ltl2dba_Q	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_U1	TIMEOUT	AI_ERROR	FALSE_RESULT
ltl2dba_alpha	TIMEOUT	NO_CODE	NO_CODE
ltl2dba_beta	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
mux	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
narylatch	TIMEOUT	NO_CODE	ERROR_CONVERT _TO_AIGER
prioritized_ arbiter	TIMEOUT	NO_CODE	FALSE_RESULT
prioritized_ arbiter_enc	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
round_robin_ arbiter	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	TIMEOUT	NO_CODE	NO_CODE
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	ERROR_COMBINE_AIGER

Table B.16.: Results for *SC-Parametric-Strix* using PaLM2 (single run)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	ERROR_COMBINE_AIGER
amba_ decomposed_ lock	AI_ERROR	NO_CODE	FALSE_RESULT
collector_v1	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
collector_v2	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
collector_v3	TIMEOUT	SUCCESS	VERIFICATION_TIMEOUT
detector	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
full_arbiter	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
load_balancer	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_C2	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
ltl2dba_E	TIMEOUT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
ltl2dba_Q	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_U1	TIMEOUT	AI_ERROR	FALSE_RESULT
ltl2dba_alpha	TIMEOUT	NO_CODE	FALSE_RESULT
ltl2dba_beta	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
mux	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
narylatch	TIMEOUT	NO_CODE	FALSE_RESULT
prioritized_ arbiter	TIMEOUT	NO_CODE	VERIFICATION_TIMEOUT
prioritized_ arbiter_enc	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
round_robin_ arbiter	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	FALSE_RESULT

Table B.17.: Results for *SC-Parametric-Strix* using PaLM2 (best-of-3)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ lock	AI_ERROR	NO_CODE	FALSE_RESULT
collector_v1	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
collector_v2	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
collector_v3	TIMEOUT	SUCCESS	VERIFICATION_TIMEOUT
detector	TIMEOUT	NO_CODE	SUCCESS
full_arbiter	NO_CODE	AI_ERROR	ERROR_CONVERT _TO_AIGER
load_balancer	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_C2	TIMEOUT	NO_CODE	NO_CODE
ltl2dba_E	TIMEOUT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
ltl2dba_Q	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_U1	TIMEOUT	AI_ERROR	FALSE_RESULT
ltl2dba_alpha	TIMEOUT	NO_CODE	ERROR_CONVERT _TO_AIGER
ltl2dba_beta	TIMEOUT	AI_ERROR	FALSE_RESULT
mux	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
narylatch	TIMEOUT	NO_CODE	ERROR_CONVERT _TO_AIGER
prioritized_ arbiter	TIMEOUT	NO_CODE	FALSE_RESULT
prioritized_ arbiter_enc	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
round_robin_ arbiter	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	FALSE_RESULT

Table B.18.: Results for *SC-Parametric-Strix* using PaLM2 (best-of-5)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	NO_CODE	NO_CODE
amba_ decomposed_ encode	AI_ERROR	NO_CODE	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ lock	AI_ERROR	NO_CODE	ERROR_CONVERT _TO_AIGER
collector_v1	NO_CODE	AI_ERROR	ERROR_CONVERT _TO_AIGER
collector_v2	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
collector_v3	TIMEOUT	ERROR_CONVERT _TO_AIGER	NO_CODE
detector	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
full_arbiter	ERROR_CONVERT _TO_AIGER	NO_CODE	ERROR_CONVERT _TO_AIGER
load_balancer	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER	NO_CODE
ltl2dba_C2	TIMEOUT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
ltl2dba_E	TIMEOUT	VERIFICATION_TIMEOUT	ERROR_COMBINE_AIGER
ltl2dba_Q	TIMEOUT	NO_CODE	ERROR_CONVERT _TO_AIGER
ltl2dba_U1	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
ltl2dba_alpha	TIMEOUT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
ltl2dba_beta	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
mux	TIMEOUT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
narylatch	TIMEOUT	NO_CODE	FALSE_RESULT
prioritized_ arbiter	TIMEOUT	NO_CODE	NO_CODE
prioritized_ arbiter_enc	TIMEOUT	NO_CODE	ERROR_CONVERT _TO_AIGER
round_robin_ arbiter	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
shift	SUCCESS	SUCCESS	NO_CODE
simple_arbiter	TIMEOUT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
simple_arbiter_ enc	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT

Table B.19.: Results for *SC-Parametric-Strix* using GPT3.5 (single run)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	NO_CODE	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ lock	AI_ERROR	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
collector_v1	FALSE_RESULT	AI_RATELIMIT	ERROR_COMBINE_AIGER
collector_v2	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
collector_v3	TIMEOUT	ERROR_CONVERT _TO_AIGER	SUCCESS
detector	TIMEOUT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
full_arbiter	ERROR_CONVERT _TO_AIGER	NO_CODE	ERROR_CONVERT _TO_AIGER
load_balancer	FALSE_RESULT	NO_CODE	ERROR_CONVERT _TO_AIGER
ltl2dba_C2	TIMEOUT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
ltl2dba_E	TIMEOUT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
ltl2dba_Q	TIMEOUT	ERROR_CONVERT _TO_AIGER	ERROR_COMBINE_AIGER
ltl2dba_U1	TIMEOUT	AI_RATELIMIT	ERROR_CONVERT _TO_AIGER
ltl2dba_alpha	TIMEOUT	ERROR_CONVERT _TO_AIGER	VERIFICATION_TIMEOUT
ltl2dba_beta	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
mux	TIMEOUT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
narylatch	TIMEOUT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
prioritized_ arbiter	TIMEOUT	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
prioritized_ arbiter_enc	TIMEOUT	NO_CODE	FALSE_RESULT
round_robin_ arbiter	TIMEOUT	AI_ERROR	ERROR_CONVERT _TO_AIGER
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	TIMEOUT	FALSE_RESULT	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	NO_CODE	FALSE_RESULT

Table B.20.: Results for *SC-Parametric-Strix* using GPT3.5 (best-of-3)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	FALSE_RESULT
amba_ decomposed_ lock	AI_ERROR	ERROR_COMBINE_AIGER	ERROR_COMBINE_AIGER
collector_v1	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
collector_v2	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
collector_v3	TIMEOUT	SUCCESS	ERROR_CONVERT _TO_AIGER
detector	TIMEOUT	NO_CODE	FALSE_RESULT
full_arbiter	NO_CODE	AI_ERROR	ERROR_COMBINE_AIGER
load_balancer	NO_CODE	AI_ERROR	ERROR_CONVERT _TO_AIGER
ltl2dba_C2	TIMEOUT	NO_CODE	ERROR_COMBINE_AIGER
ltl2dba_E	TIMEOUT	FALSE_RESULT	ERROR_COMBINE_AIGER
ltl2dba_Q	TIMEOUT	ERROR_COMBINE_AIGER	ERROR_COMBINE_AIGER
ltl2dba_U1	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
ltl2dba_alpha	TIMEOUT	NO_CODE	ERROR_CONVERT _TO_AIGER
ltl2dba_beta	TIMEOUT	AI_ERROR	FALSE_RESULT
mux	TIMEOUT	FALSE_RESULT	FALSE_RESULT
narylatch	TIMEOUT	AI_ERROR	FALSE_RESULT
prioritized_ arbiter	TIMEOUT	NO_CODE	FALSE_RESULT
prioritized_ arbiter_enc	TIMEOUT	AI_ERROR	ERROR_COMBINE_AIGER
round_robin_ arbiter	TIMEOUT	AI_ERROR	FALSE_RESULT
shift	SUCCESS	SUCCESS	FALSE_RESULT
simple_arbiter	TIMEOUT	NO_CODE	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	ERROR_CONVERT _TO_AIGER

Table B.21.: Results for *SC-Parametric-Strix* using GPT3.5 (best-of-5)

B.2.2. Comparison of Different Verilog Translations with BoSy

A comparison of how the different translation methods compare.

- **standard**: The standard Verilog output option BoSy provides.

- **aag**: Directly translating the Aiger output from BoSy.
- **opt_aag**: As above, but before translating an optimisation pass is done with Yosys.
- **opt_verilog**: The BoSy verilog output is first translated into Aiger, then optimised using Yosys, and then translated back into Verilog.

Benchmark	aag	opt_aag	opt_verilog	standard
amba_ decomposed_ arbiter	NO_CODE	NO_CODE	NO_CODE	NO_CODE
amba_ decomposed_ encode	NO_CODE	NO_CODE	NO_CODE	NO_CODE
amba_ decomposed_ lock	NO_CODE	NO_CODE	NO_CODE	NO_CODE
collector_v2	ERROR_CONVERT _TO_AIGER	NO_CODE	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
collector_v3	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT	SUCCESS
detector	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
load_balancer	NO_CODE	NO_CODE	NO_CODE	NO_CODE
ltl2dba_C2	FALSE_RESULT	NO_CODE	NO_CODE	FALSE_RESULT
ltl2dba_alpha	FALSE_RESULT	FALSE_RESULT	NO_CODE	FALSE_RESULT
mux	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
narylatch	NO_CODE	NO_CODE	NO_CODE	NO_CODE
prioritized_ arbiter	ERROR_CONVERT _TO_AIGER	NO_CODE	NO_CODE	FALSE_RESULT
shift	FALSE_RESULT	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
simple_arbiter_ enc	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER	FALSE_RESULT

Table B.22.: PaLM2

Benchmark	aag	opt_aag	opt_verilog	standard
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	AI_ERROR	AI_ERROR
amba_ decomposed_ encode	AI_ERROR	AI_ERROR	AI_ERROR	AI_ERROR
amba_ decomposed_ lock	NO_CODE	NO_CODE	AI_ERROR	AI_ERROR
collector_v2	NO_CODE	ERROR_CONVERT _TO_AIGER	NO_CODE	NO_CODE
collector_v3	SUCCESS	SUCCESS	SUCCESS	SUCCESS
detector	NO_CODE	NO_CODE	NO_CODE	NO_CODE
load_balancer	NO_CODE	NO_CODE	NO_CODE	ERROR_CONVERT _TO_AIGER
ltl2dba_C2	ERROR_CONVERT _TO_AIGER	NO_CODE	ERROR_CONVERT _TO_AIGER	NO_CODE
ltl2dba_alpha	NO_CODE	NO_CODE	NO_CODE	NO_CODE
mux	NO_CODE	ERROR_CONVERT _TO_AIGER	NO_CODE	SUCCESS
narylatch	AI_ERROR	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
prioritized_ arbiter	NO_CODE	NO_CODE	NO_CODE	FALSE_RESULT
shift	ERROR_CONVERT _TO_AIGER	SUCCESS	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
simple_arbiter	NO_CODE	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	SUCCESS
simple_arbiter_ enc	NO_CODE	NO_CODE	NO_CODE	NO_CODE

Table B.23.: GPT3.5

B.2.3. Scaling Solutions to their Highest Parameter Values

Benchmark	Self	BoSy	Strix	None
detector	8	0	0	0
full_arbiter	4	0	0	0
mux	128	128	0	128
shift	512	128	128	128
simple_arbiter	16	16	0	0

Table B.24.: Highest parameter value for which a correct solution was produced by GPT3.5 on *SC-Parametric-Human*

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	0	0	0
amba_ decomposed_ encode	0	0	0
amba_ decomposed_ lock	0	0	0
collector_v2	0	0	0
collector_v3	128	0	0
detector	0	0	0
load_balancer	0	0	0
ltl2dba_C2	0	0	0
ltl2dba_alpha	0	0	0
mux	0	0	64
narylatch	0	0	0
prioritized_ arbiter	0	0	0
shift	64	256	0
simple_arbiter	16	0	0
simple_arbiter_ enc	0	0	0

Table B.25.: Highest parameter value for which a correct solution was produced by GPT3.5 on *SC-Parametric-BoSy*

B.2.4. One-shot Results

Benchmark	Self	BoSy	Strix
detector	SUCCESS	FALSE_RESULT	NO_CODE
full_arbiter	ERROR_COMBINE_AIGER	NO_CODE	NO_CODE
mux	SUCCESS	FALSE_RESULT	SUCCESS
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	SUCCESS	FALSE_RESULT

Table B.26.: One-shot results for *SC-Parametric-Human* using PaLM2 (best-of-3)

Benchmark	Self	BoSy	Strix
detector	SUCCESS	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
full_arbiter	SUCCESS	FALSE_RESULT	NO_CODE
mux	SUCCESS	ERROR_CONVERT _TO_AIGER	ERROR_CONVERT _TO_AIGER
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	SUCCESS	FALSE_RESULT

Table B.27.: One-shot results for *SC-Parametric-Human* using GPT3.5 (best-of-3)

Benchmark	BoSy	Strix
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR
amba_ decomposed_ encode	AI_ERROR	NO_CODE
amba_ decomposed_ lock	AI_ERROR	NO_CODE
collector_v2	AI_ERROR	AI_ERROR
collector_v3	SUCCESS	SUCCESS
detector	FALSE_RESULT	NO_CODE
load_balancer	NO_CODE	AI_ERROR
ltl2dba_C2	FALSE_RESULT	NO_CODE
ltl2dba_alpha	FALSE_RESULT	FALSE_RESULT
mux	FALSE_RESULT	SUCCESS
narylatch	AI_ERROR	NO_CODE
prioritized_ arbiter	FALSE_RESULT	NO_CODE
shift	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR

Table B.28.: One-shot results for *SC-Parametric-BoSy* using PaLM2 (best-of-3)

Benchmark	BoSy	Strix
amba_ decomposed_ arbiter	AI_ERROR	NO_CODE
amba_ decomposed_ encode	AI_ERROR	ERROR_CONVERT_TO_AIGER
amba_ decomposed_ lock	AI_ERROR	ERROR_CONVERT_TO_AIGER
collector_v2	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
collector_v3	SUCCESS	SUCCESS
detector	FALSE_RESULT	FALSE_RESULT
load_balancer	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
ltl2dba_C2	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_alpha	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
mux	SUCCESS	FALSE_RESULT
narylatch	AI_ERROR	ERROR_CONVERT_TO_AIGER
prioritized_ arbiter	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
shift	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	ERROR_COMBINE_AIGER

Table B.29.: One-shot results for *SC-Parametric-BoSy* using GPT3.5 (best-of-3)

Benchmark	BoSy	Strix
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR
amba_ decomposed_ encode	AI_ERROR	NO_CODE
amba_ decomposed_ lock	AI_ERROR	NO_CODE
collector_v1	ERROR_CONVERT_TO_AIGER	AI_ERROR
collector_v2	TIMEOUT	AI_ERROR
collector_v3	TIMEOUT	SUCCESS
detector	TIMEOUT	NO_CODE
full_arbiter	NO_CODE	AI_ERROR
load_balancer	NO_CODE	AI_ERROR
ltl2dba_C2	TIMEOUT	NO_CODE
ltl2dba_E	TIMEOUT	ERROR_CONVERT_TO_AIGER
ltl2dba_Q	TIMEOUT	NO_CODE
ltl2dba_U1	TIMEOUT	AI_ERROR
ltl2dba_alpha	TIMEOUT	VERIFICATION_TIMEOUT
ltl2dba_beta	TIMEOUT	AI_ERROR
mux	TIMEOUT	SUCCESS
narylatch	TIMEOUT	NO_CODE
prioritized_ arbiter	TIMEOUT	NO_CODE
prioritized_ arbiter_enc	TIMEOUT	AI_ERROR
round_robin_ arbiter	TIMEOUT	AI_ERROR
shift	SUCCESS	SUCCESS
simple_arbiter	TIMEOUT	NO_CODE
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR

Table B.30.: One-shot results for *SC-Parametric-Strix* using PaLM2 (best-of-3)

Benchmark	BoSy	Strix
amba_ decomposed_ arbiter	AI_ERROR	ERROR_CONVERT_TO_AIGER
amba_ decomposed_ encode	AI_ERROR	NO_CODE
amba_ decomposed_ lock	AI_ERROR	ERROR_CONVERT_TO_AIGER
collector_v1	ERROR_CONVERT_TO_AIGER	AI_RATELIMIT
collector_v2	TIMEOUT	AI_ERROR
collector_v3	TIMEOUT	SUCCESS
detector	TIMEOUT	FALSE_RESULT
full_arbiter	FALSE_RESULT	NO_CODE
load_balancer	ERROR_CONVERT_TO_AIGER	NO_CODE
ltl2dba_C2	TIMEOUT	ERROR_CONVERT_TO_AIGER
ltl2dba_E	TIMEOUT	ERROR_CONVERT_TO_AIGER
ltl2dba_Q	TIMEOUT	ERROR_CONVERT_TO_AIGER
ltl2dba_U1	TIMEOUT	AI_RATELIMIT
ltl2dba_alpha	TIMEOUT	ERROR_CONVERT_TO_AIGER
ltl2dba_beta	TIMEOUT	AI_ERROR
mux	TIMEOUT	ERROR_CONVERT_TO_AIGER
narylatch	TIMEOUT	ERROR_CONVERT_TO_AIGER
prioritized_ arbiter	TIMEOUT	FALSE_RESULT
prioritized_ arbiter_enc	TIMEOUT	NO_CODE
round_robin_ arbiter	TIMEOUT	AI_ERROR
shift	SUCCESS	SUCCESS
simple_arbiter	TIMEOUT	FALSE_RESULT
simple_arbiter_ enc	FALSE_RESULT	ERROR_CONVERT_TO_AIGER

Table B.31.: One-shot results for *SC-Parametric-Strix* using GPT3.5 (best-of-3)

B.2.5. Zero-shot Results with the Prompt Containing the Module Definition

All experiments here were performed as a best-of-3 using the prompt described in Sect. 4.5.1. Because we are only listing zero-shot results, the columns in these tables contain the LLM which was used. Here we only evaluated each benchmark using the reference solution provided by its corresponding tool (e.g. *SC-Parametric-BoSy* only uses BoSy for its examples).

→ Sec. 4.5.1, p. 25

Benchmark	GPT3.5	PaLM2
detector	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
full_arbiter	ERROR_CONVERT_TO_AIGER	FALSE_RESULT
mux	SUCCESS	SUCCESS
shift	SUCCESS	SUCCESS
simple_arbiter	NO_CODE	FALSE_RESULT

Table B.32.: Zero-shot results for *SC-Parametric-Human* using the prompt containing the module definition for (best-of-3)

Benchmark	GPT3.5	PaLM2
amba_ decomposed_ arbiter	ERROR_CONVERT_TO_AIGER	ERROR_COMBINE_AIGER
amba_ decomposed_ encode	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
amba_ decomposed_ lock	ERROR_CONVERT_TO_AIGER	FALSE_RESULT
collector_v2	FALSE_RESULT	FALSE_RESULT
collector_v3	SUCCESS	SUCCESS
detector	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
load_balancer	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_C2	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_alpha	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
mux	SUCCESS	SUCCESS
narylatch	FALSE_RESULT	FALSE_RESULT
prioritized_ arbiter	FALSE_RESULT	ERROR_COMBINE_AIGER
shift	SUCCESS	SUCCESS
simple_arbiter	ERROR_CONVERT_TO_AIGER	NO_CODE
simple_arbiter_ enc	FALSE_RESULT	ERROR_CONVERT_TO_AIGER

Table B.33.: Zero-shot results for *SC-Parametric-BoSy* using the prompt containing the module definition for (best-of-3)

Benchmark	GPT3.5	PaLM2
amba_ decomposed_ arbiter	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
amba_ decomposed_ encode	FALSE_RESULT	FALSE_RESULT
amba_ decomposed_ lock	FALSE_RESULT	FALSE_RESULT
collector_v1	FALSE_RESULT	FALSE_RESULT
collector_v2	FALSE_RESULT	FALSE_RESULT
collector_v3	SUCCESS	ERROR_CONVERT_TO_AIGER
detector	ERROR_CONVERT_TO_AIGER	NO_CODE
full_arbiter	FALSE_RESULT	FALSE_RESULT
load_balancer	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_C2	FALSE_RESULT	NO_CODE
ltl2dba_E	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_Q	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_U1	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_alpha	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_beta	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
mux	SUCCESS	SUCCESS
narylatch	FALSE_RESULT	FALSE_RESULT
prioritized_ arbiter	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
prioritized_ arbiter_enc	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
round_robin_ arbiter	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
shift	SUCCESS	SUCCESS
simple_arbiter	FALSE_RESULT	NO_CODE
simple_arbiter_ enc	FALSE_RESULT	ERROR_CONVERT_TO_AIGER

Table B.34.: Zero-shot results for *SC-Parametric-Strix* using the prompt containing the module definition for (best-of-3)

B.2.6. Initial GPT-4 results

These are the results from initial experiments we performed using GPT-4.

Benchmark	Self	BoSy	Strix	None
detector	SUCCESS	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
full_arbiter	ERROR_COMBINE _AIGER	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
mux	SUCCESS	SUCCESS	SUCCESS	SUCCESS
shift	SUCCESS	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	SUCCESS	SUCCESS	NO_CODE

Table B.35.: Results for *SC-Parametric-Human* using GPT-4 (best-of-3)

Benchmark	BoSy	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	AI_ERROR	NO_CODE
amba_ decomposed_ encode	AI_RATELIMIT	AI_ERROR	FALSE_RESULT
amba_ decomposed_ lock	AI_ERROR	AI_ERROR	ERROR_CONVERT _TO_AIGER
collector_v2	FALSE_RESULT	ERROR_CONVERT _TO_AIGER	FALSE_RESULT
collector_v3	SUCCESS	SUCCESS	SUCCESS
detector	FALSE_RESULT	FALSE_RESULT	FALSE_RESULT
load_balancer	FALSE_RESULT	AI_ERROR	FALSE_RESULT
ltl2dba_C2	SUCCESS	FALSE_RESULT	SUCCESS
ltl2dba_alpha	FALSE_RESULT	FALSE_RESULT	ERROR_CONVERT _TO_AIGER
mux	SUCCESS	SUCCESS	SUCCESS
narylatch	AI_RATELIMIT	SUCCESS	FALSE_RESULT
prioritized_ arbiter	FALSE_RESULT	FALSE_RESULT	NO_CODE
shift	SUCCESS	SUCCESS	SUCCESS
simple_arbiter	SUCCESS	SUCCESS	ERROR_COMBINE_AIGER
simple_arbiter_ enc	FALSE_RESULT	AI_ERROR	FALSE_RESULT

Table B.36.: Results for *SC-Parametric-BoSy* using GPT-4 (best-of-3)

Benchmark	Strix	None
amba_ decomposed_ arbiter	AI_ERROR	ERROR_CONVERT_TO_AIGER
amba_ decomposed_ encode	AI_ERROR	FALSE_RESULT
amba_ decomposed_ lock	AI_ERROR	ERROR_CONVERT_TO_AIGER
collector_v1	AI_RATELIMIT	ERROR_CONVERT_TO_AIGER
collector_v2	AI_RATELIMIT	ERROR_CONVERT_TO_AIGER
collector_v3	SUCCESS	ERROR_CONVERT_TO_AIGER
detector	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
full_arbiter	AI_ERROR	FALSE_RESULT
load_balancer	AI_ERROR	FALSE_RESULT
ltl2dba_C2	FALSE_RESULT	VERIFICATION_TIMEOUT
ltl2dba_E	FALSE_RESULT	ERROR_CONVERT_TO_AIGER
ltl2dba_Q	NO_CODE	ERROR_CONVERT_TO_AIGER
ltl2dba_U1	AI_RATELIMIT	ERROR_CONVERT_TO_AIGER
ltl2dba_alpha	ERROR_CONVERT_TO_AIGER	ERROR_CONVERT_TO_AIGER
ltl2dba_beta	AI_RATELIMIT	ERROR_CONVERT_TO_AIGER
mux	SUCCESS	SUCCESS
narylatch	AI_ERROR	FALSE_RESULT
prioritized_ arbiter	NO_CODE	ERROR_CONVERT_TO_AIGER
prioritized_ arbiter_enc	AI_ERROR	FALSE_RESULT
round_robin_ arbiter	AI_RATELIMIT	FALSE_RESULT
shift	SUCCESS	SUCCESS
simple_arbiter	VERIFICATION_TIMEOUT	FALSE_RESULT
simple_arbiter_ enc	AI_ERROR	FALSE_RESULT

Table B.37.: Results for *SC-Parametric-Strix* using GPT-4 (best-of-3)

Appendix C

Prompts Used

C.1. PaLM2 Prompt

In the below figure you can see the default prompt given to PaLM2. Note that it only differs from the GPT3.5 prompt in the way that the examples are presented to the model. Instead of being part of the message history, they are structurally separated.

System Example 1

USER Please write a Verilog module for n=2 fulfilling the following specification. Make sure the code is fully synthesizable.:

```
G (F r_0) && G (F r_1) <-> G (F g)
```

ASSISTANT

```
module detector(r_0, r_1, g);
...
endmodule
```

System Example 2

USER Please write a Verilog module for n=4 fulfilling the following specification. Make sure the code is fully synthesizable.:

```
G (F r_0) && G (F r_1) && G (F r_2) && G (F r_3) <-> G (F g)
```

ASSISTANT

```
module detector(r_0, r_1, r_2, r_3, g);
...
endmodule
```

Prompt

USER Please write a Verilog module for n=8 fulfilling the following specification. Make sure the code is fully synthesizable.:

```
G (F r_0) && G (F r_1) && G (F r_2) && G (F r_3) && G (F r_4) && G (F r_5) && G
(F r_6) && G (F r_7) <-> G (F g)
```

Figure C.1.: The prompt used for PaLM2. Terminology slightly adjusted for consistency.

C.2. Module Definition Prompt

➔Sec. 4.5.1, p. 25 This is the modified zero-shot prompt used in Sect. 4.5.1. For the sake of brevity we will not distinguish between PaLM2 and GPT3.5 here.

You are an expert in writing correct Verilog code, that fulfill certain formal properties specified in LTL.
Please write a Verilog module for n=4 using the following module definition as a basis.

```
module detector(r_0, r_1, r_2, r_3, g)
```

The code needs to be fully synthesizable and follow the following LTL specification:
`G (F r_0) && G (F r_1) && G (F r_2) && G (F r_3) <-> G (F g)`

Figure C.2.: The zero-shot prompt additionally containing the module definitifon