Preliminary Experiments using LLMs for Design

John T Maxwell III, **Johan de Kleer**, **Ion Matei**, **Maksym Zhenirovsky**
Palo Alto Research Center
SRI International
{maxwell, dekleer, matei, mazhenir}@parc.com

Abstract

In analogy to using an LLM to generate a story on some topic, or Dall-E to generate an image, we can use LLMs to design a physical system to achieve a function. A system such as ChatGPT isn't a great designer, but it has two significant advantages. First, the designs it produces are approximately correct and thus we can use it as a starting point for developing a practical design. Second, it has a vast amount of knowledge about physical domains and is not limited to one domain. All other design tools have severe domain limitations. In short, it is an extremely general but sloppy designer. In this paper we show how the sloppy designs produced by ChatGPT can refined to produce practical designs.

1 Design experiment methodology

For all our design experiments we start with a known design (e.g., low-pass filter, power-train, op-amp) and construct a data set by simulating its behaviors over time that characterize its function. For example, for a low-pass filter we simulate its behavior at a frequency within its pass-band, and outside of it. We provide those input/output sequences to our automated designer to construct a system which produces the same input/output sequences. There are often multiple ways of achieving the same input-output behavior, so the designed system may not have the same topology or parameters as our original system. We are not trying to recreate the original design, but rather to automatically construct a design which behaves in the same way. There are usually an infinite number of ways a desired function can be achieved. Our approach finds a simpler one simply because the LLM [Devlin *et al.*, 2019] will typically find simpler ones.

An immediate challenge to using ChatGPT [Roumeliotis and Tselikas, 2023] is that it has a poor grasp of mathematics. Hence, we use ChatGPT only to generate a topology of components. ChatGPT cannot assign parameter values to components. That task is left to an optimizer which picks values for component parameters such that the function of the system is achieved.

2 Low-pass filter

Consider designing a low pass filter. A low pass filter is a circuit which reduces the high frequencies in the input signal and passes through unaltered low frequencies (hence the name). Figure 1 is a simple example of a low-pass filter. To construct the input-output data set let $R = 1K\Omega$ and $C = 0.1\mu F$. With those values the circuit has a cutoff frequency of approximately $1.6KHz$ (the frequency at which there is $3db$ loss in amplitude. To frame our experiment we simulate our circuit at $f = 1KHz$ and $f = 2KHz$. Assume the load impedance is $10K\Omega$. We then simulate the low-pass filter to construct two sequences of input/outputs. The output voltage time series corresponding to a sinusoidal input at $1KHz$ and $2KHz$ frequencies are shown in Figure 2. These time series will serve as ground truth for the design challenge.

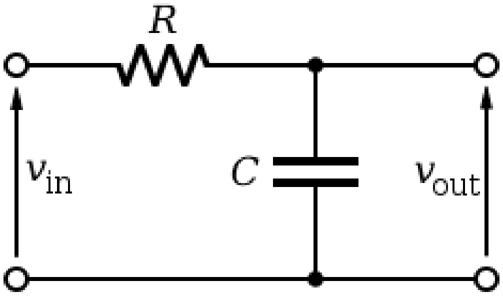


Figure 1: Low pass filter

3 Modelica and ChatGPT

We use Modelica [Fritzson, 2004] as our primary modelling tool. Modelica consists of a modeling language and a simulator so that we can test any designs that are discovered. The ChatGPT training set includes enough Modelica models that we can use ChatGPT 4.0 directly. We first construct a prompt which includes the library of Modelica components to choose among, the interface of the desired system, and the natural language description of the desired function. In order to design the low-pass filter we provide ChatGPT 4.0 the following prompt:

```
Here is a library of Modelica components:
```

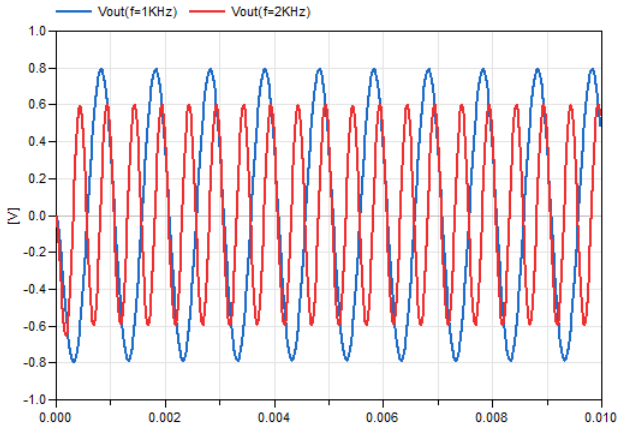


Figure 2: Output voltage for a sinusoidal input voltage $1KHz$ and $2KHz$ frequencies.

```
model Modelica.Electrical.Analog.Basic.Capacitor
  Modelica.Electrical.Analog.Interfaces.PositivePin p;
  Modelica.Electrical.Analog.Interfaces.NegativePin n;
end Modelica.Electrical.Analog.Basic.Capacitor;

model Modelica.Electrical.Analog.Basic.Ground
  Modelica.Electrical.Analog.Interfaces.Pin p;
end Modelica.Electrical.Analog.Basic.Ground;

model Modelica.Electrical.Analog.Basic.Inductor
  Modelica.Electrical.Analog.Interfaces.PositivePin p;
  Modelica.Electrical.Analog.Interfaces.NegativePin n;
end Modelica.Electrical.Analog.Basic.Inductor;

model Modelica.Electrical.Analog.Basic.Resistor
  Modelica.Electrical.Analog.Interfaces.PositivePin p;
  Modelica.Electrical.Analog.Interfaces.NegativePin n;
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a heatPort;
end Modelica.Electrical.Analog.Basic.Resistor;

model Modelica.Electrical.Analog.Ideal.IdealOpAmp3Pin
  Modelica.Electrical.Analog.Interfaces.PositivePin in_p;
  Modelica.Electrical.Analog.Interfaces.NegativePin in_n;
  Modelica.Electrical.Analog.Interfaces.PositivePin out;
end Modelica.Electrical.Analog.Ideal.IdealOpAmp3Pin;

Here is the interface for a low pass filter:

model two_port_interface
Modelica.Electrical.Analog.Interfaces.PositivePin p_in;
Modelica.Electrical.Analog.Interfaces.NegativePin n_in;
Modelica.Electrical.Analog.Interfaces.PositivePin p_out;
Modelica.Electrical.Analog.Interfaces.NegativePin n_out;
end two_port_interface;

Connect components from the library into a well-formed
Modelica model to create a low pass filter that extends
two_port_interface.  Name the Modelica model ``low_pass_filter''.
```

ChatGPT outputs the following Modelica code:

```
model low_pass_filter "Low Pass Filter"
  extends step_function_test;

  Modelica.Electrical.Analog.Basic.Resistor
     R1(R=1000);
  Modelica.Electrical.Analog.Basic.Capacitor
     C1(C=1e-6);

equation
  connect(p_in, R1.p);
  connect(R1.n, C1.p);
  connect(R1.n, p_out);
  connect(C1.n, n_out);
end low_pass_filter;
```

We will address the fact that the indicated parameters are incorrect for the task. ChatGPT cannot find correct parameters even if they exist because it understands so little about equations. We have built a simple automatic renderer for Modelica models, Figure 3 is drawn by our renderer, not ChatGPT. Hence, the drawings appear awkward.

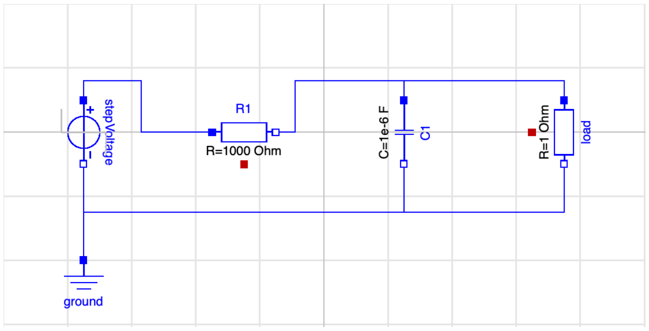


Figure 3: Low pass filter

4 Generate and Test

Figure 4 illustrates our approach to using ChatGPT in design. Once we have a Modelica model topology, we use the Sizer to determine the values of the parameters so that the input/output sequences match as closely as possible (through optimization).

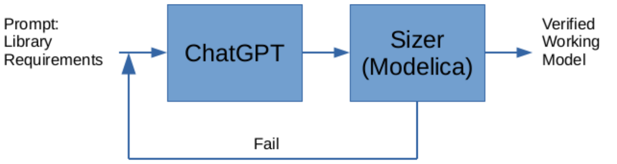


Figure 4: Ideal Framework: ChatGPT constructs a Modelica topology from the requirements and and optimization is used to determine the exact parameters needed. If no parameters can be found for that topology that achieve the desired function, the generate and test loop continues.

We use Dymola to convert the Modelica model into an FMU. We then use a gradient-free descent method (Powell) to determine the best values for the parameters. We optimize the parameters against the desired input-output function data set. Thus we use ChatGPT to determine a topological model from specifications written in English, and continuous optimization to determine the parameters of the topology that match the requirements. In the case of the RC circuit, only the product RC is relevant, thus the optimizer can find various values for R and C, depending on the initial conditions of the algorithm. For example for one run with random initial conditions in $[0, 1]$ and $1KHz$ input frequency, the Sizer finds $R = 0.00969\Omega$, $C = 0.01041F$. Their product is roughly 10^{-4}, which is exactly the product of the R, C parameters for the ground truth case.

Note however that ChatGPT will not always generate topologies for which the Sizer can find any appropriate values for the system parameters. For example, in the case of a purely resistive circuit, the Sizer will never find correct parameters that can match the ground truth. Worse, the models ChatGPT constructs can be syntactically incorrect. These errors occur because LLMs are not general AIs and make many mistakes. The surprise is more than ChatGPT can often find almost correct designs.

Since ChatGPT is stochastic, every invocation of ChatGPT often yields a new topology. Therefore, one could just keep on calling ChatGPT over and over again until, hopefully, it comes up with a valid topology. This obviously yields very poor performance. Figures 5 and 6 show two faulty topologies ChatGPT generates.

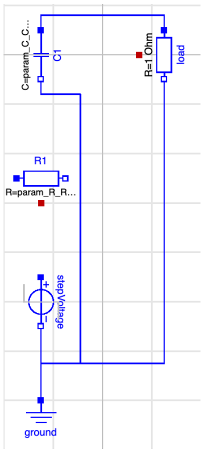


Figure 5: Bad low pass filter. This has a disconnected input and its output is always 0. It attenuates all frequences.

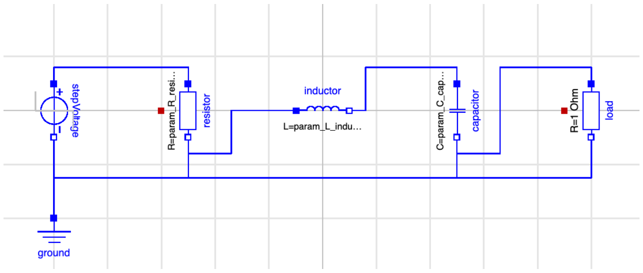


Figure 6: Another bad low pass filter. This one is syntactically correct, but output is always 0. It attenuates all frequences.

5 Generate and Repair

Instead of taking a pure "generate and test" approach, we could try to repair the designs that are syntactically invalid or that do not have the proper behavior.

One way to try to repair the designs would be to ask ChatGPT to redesign a faulty design given some information of what was wrong with it. We did some experiments to try this out. Since ChatGPT is stochastic, it was hard to tell whether the new information made a difference. So we tried each repair ten times both with and without the new information. For a few repairs it was more likely to produce a good design with the new information, but for most repairs there was no difference. In this case, using ChatGPT for repair was effectively a "generate and test" approach.

To perform our experiments we introduce a 3rd module to our framework. The repair module detects whether there is some simple syntactic reason the Modelica topology produced by ChatGPT cannot function. (It also checks for duplicate topologies.) It then attempts to repair the topology with a a simple local transformation. This is much less expensive than starting the Modelica optimizer. Figure 7 is the framework we use in our experiments.

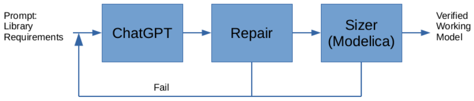


Figure 7: More efficient framework which repairs syntactically incorrect Modelica topologies.

Although we could have the repair module generate a sequence of designs to test, we instead have the repair module generate a model that represents a space of designs where the different design choices are represented by switches. Currently, we generate a sequence of designs to test from the space of designs, but in the future we hope to use ATMS[de Kleer, 1986] reasoning to search the space more efficiently than by using exhaustive enumeration.

The first thing that the repair module does is repair syntactically invalid models. For instance, ChatGPT sometimes leaves out the 'equation' keyword from the model. The repair module detects this and inserts an 'equation' keyword between the components and the connections.

The next thing the repair module does is to look for possible topological errors such as disconnected ports (Figure 5) and positive ports connected to ground (Figure 6). For each possible error, it generates a range of alternatives. For instance, it converts Figure 5 into Figure 8, and Figure 6 into Figure 9.

Once we have a repair space, we can generate candidate designs by enumerating switch values that are consistent with the 'oneof' constraints and passing the resulting design to the Sizer to determine optimal parameter values. Figures 8 and 9 contain valid low-pass filter designs, so this process produces a successful design in these cases.

6 Design of a Power Train

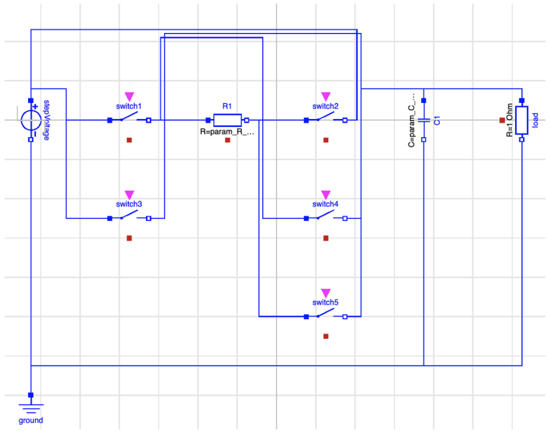
The Modelica Standard Library (MSL) contains an extensive collection of power train components. Figure 10 illustrates a simple vehicle power train. This model has a simple model of the road and driver.

The input-output function we desire is produced by a simulated driver (Figure 11).

The following ChatGPT prompt generates powertrain topologies.

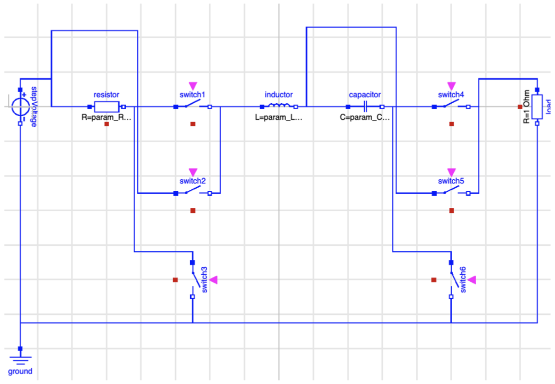
```
Here is a library of Modelica components:

model Modelica.Mechanics.Rotational.Components.Brake
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a;
```



```
oneof(switch1,switch2,switch3)
oneof(switch4,switch1)
oneof(switch5,switch2)
```

Figure 8: Repair space for Figure 5.



```
oneof(switch1,switch2) oneof(switch1,switch3)
oneof(switch4,switch5) oneof(switch4,switch6)
```

Figure 9: Repair space for Figure 6.

```
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b;
  Modelica.Blocks.Interfaces.RealInput f_normalized;
end Modelica.Mechanics.Rotational.Components.Brake;

model Powertrain.Components.Chassis
  Modelica.Blocks.Interfaces.RealOutput speed;
  Modelica.Mechanics.Translational.Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Translational.Interfaces.Flange_a flange_b;
  Modelica.Blocks.Interfaces.RealInput grade;
end Powertrain.Components.Chassis;

model Modelica.Mechanics.Rotational.Components.Clutch
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b;
  Modelica.Blocks.Interfaces.RealInput f_normalized;
end Modelica.Mechanics.Rotational.Components.Clutch;

model Powertrain.Driver.Driver
  Modelica.Blocks.Interfaces.RealInput actualSpeed;
  Modelica.Blocks.Interfaces.RealOutput gear;
  Modelica.Blocks.Interfaces.RealOutput clutch;
  Modelica.Blocks.Interfaces.RealOutput gasPedal;
  Modelica.Blocks.Interfaces.RealOutput brakePedal;
end Powertrain.Driver.Driver;
```

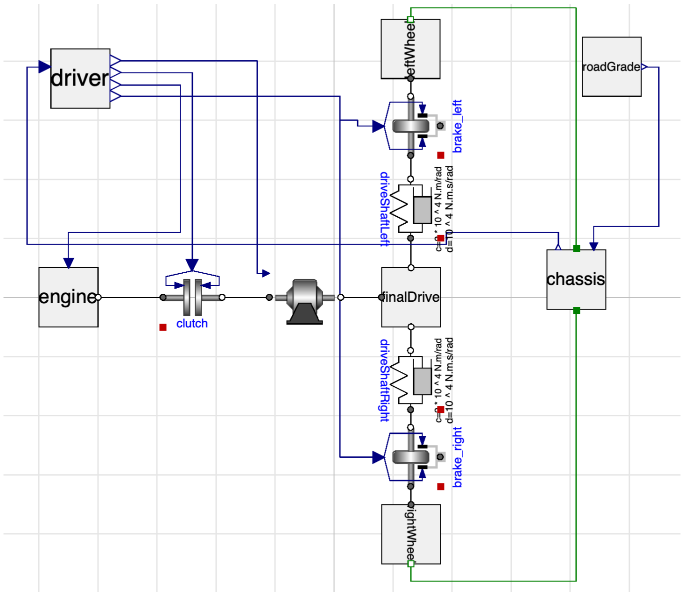


Figure 10: A working powertrain

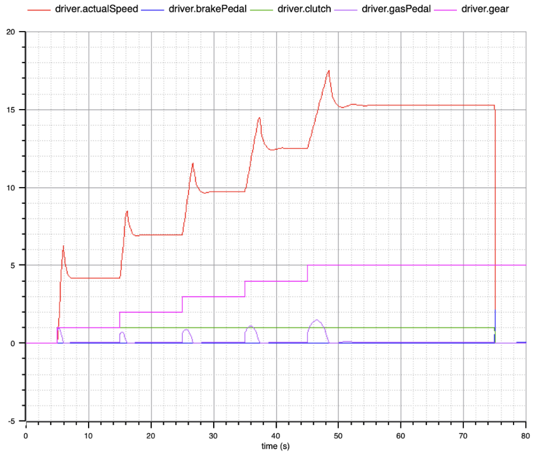


Figure 11: Desired power train behavior

```
model Powertrain.Components.Engine
  Modelica.Blocks.Interfaces.RealInput pedal;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_a;
end Powertrain.Components.Engine;

model Powertrain.Components.GearBox
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b;
  Modelica.Blocks.Interfaces.RealInput gear;
end Powertrain.Components.GearBox;

model Powertrain.Driver.RoadGrade
  Modelica.Blocks.Interfaces.RealOutput grade;
end Powertrain.Driver.RoadGrade;

model Powertrain.Components.SimplifiedwheelRoad
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Translational.Interfaces.Flange_b flange_a1;
end Powertrain.Components.SimplifiedwheelRoad;

Connect components from this library in a well-formed Modelica model
to create a drive train.
```

7 Generate and Behavioral Repair

ChatGPT 4.0 mostly generates bad topologies which are not repairable with the syntactic techniques just outlined. The Sizer does not find parameters which yield a correct function for any of the repairs. Three such bad designs are: Figures 12, 13, and 14.

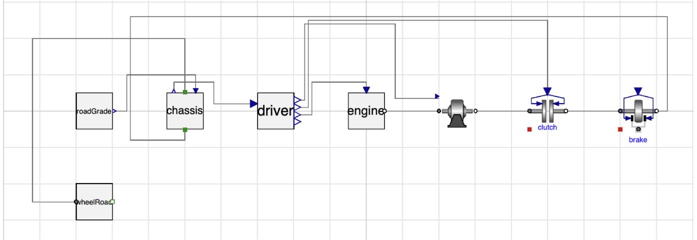


Figure 12: Problems: Brake output should be connected to wheelRoad, not chassis. Rotational flanges should not be connected to translational flanges. Missing a connection between driver.brakePedal and brake input.

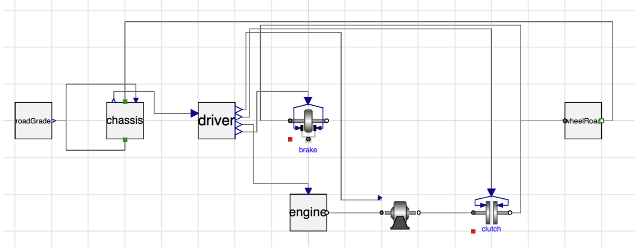


Figure 13: Problems: Brake input is connected to wheelRoad instead of output. Clutch is connected to wheelRoad instead of brake input. roadGrade is connected to chassis.flange_b (wrong type)

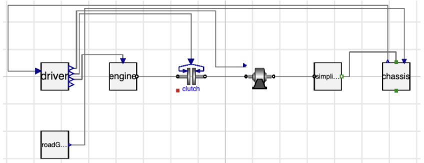


Figure 14: Problem: Missing brake

Consider the design in Figure 14. There is nothing in the design structure to indicate that the brake is missing. The problem only shows up in the behavior. In the ideal behavior, we see that when the brake pedal is pressed, the vehicle sharply decelerates (see Figure 11). This can be seen more clearly in Figure 15, where the extraneous variables have been removed.

Figure 14 doesn't use the brake pedal, so that is a clue. But what should the brake pedal be connected to? How do we know to add a brake?

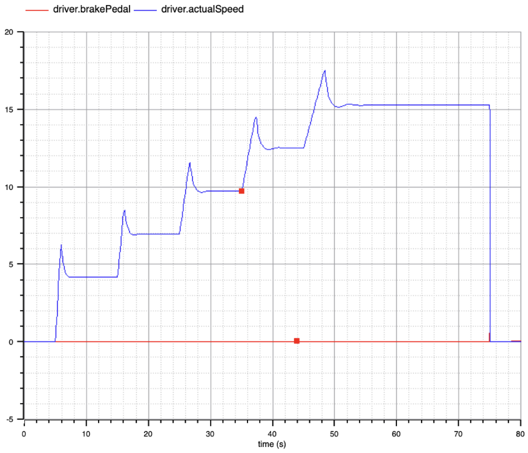


Figure 15: Behavior of brake pedal in power train

One way to determine that we are missing a brake is to look at the qualitative behavior of the brake pedal versus the speed in Figure 15. When the brake pedal is zero, then the speed can be positive or zero, and the first derivative can be positive, negative, or zero. However, when the brake pedal is positive, then the speed is always decelerating. So we can look for a component in our library that has that behavior.

Each component in the library has a unit test that exhibits the behavior of that component. The unit test for the brake is shown in Figure 16 and its behavior is shown in Figure 17.

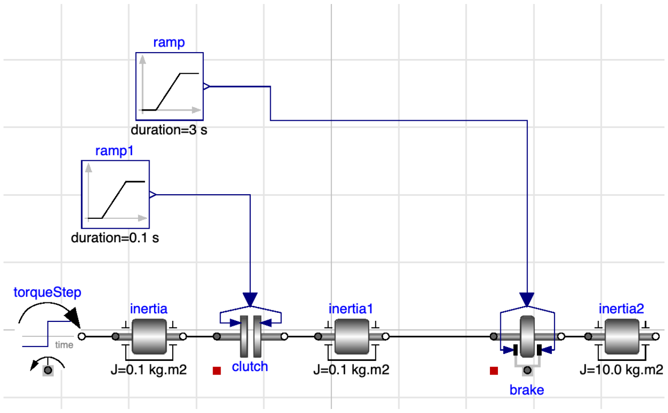


Figure 16: Unit test model for brake

Although it is not easy to see in Figure 17, the output variable starts decelerating when the input variable is (very slightly) positive. So this suggests that adding a brake might repair Figure 14. The first step to adding Figure 16 to Figure 14 is to convert the design by replacing the unit test scaffolding with the power train scaffolding. This produces Figure 18.

The next step is to merge Figure 18 and Figure 14. There are many ways to merge these two designs. If we assume that linear designs tend to remain linear, and that duplicate components are shared, then we can use code to zipper the

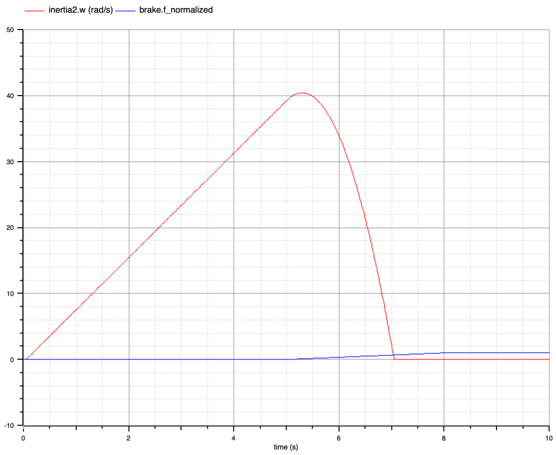


Figure 17: Behavior of brake unit test

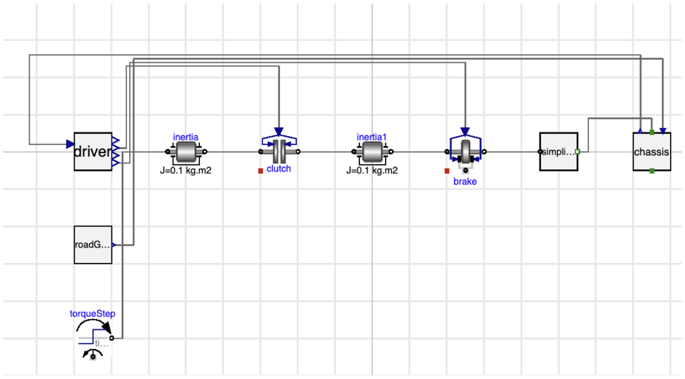


Figure 18: Brake unit test converted to power train

designs together to produce the power train in Figure 19.

We can also ask ChatGPT to merge the two designs through the following prompt:

```
Here are two Modelica models:

within Powertrain;
model vehicle1
  Powertrain.Driver.RoadGrade roadGrade;
  Powertrain.Driver.Driver driver;
  Powertrain.Components.Engine engine;
  Modelica.Mechanics.Rotational.Components.Clutch
    clutch (mue_pos=[0, 0.3], peak=1.1, cgeo=0.176, fn_max=6973);
  Powertrain.Components.SimplifiedwheelRoad
      simplifiedwheel_road(vehicle_mass=1520, radius=0.3);
  Powertrain.Components.Chassis chassis(vehicleMass=1520);
  Modelica.Mechanics.Rotational.Components.Brake
      brake(locked(fixed=true,start=true),
                mue_pos=[0,0.3], fn_max=293200.0);

equation
  connect(engine.flange_a, clutch.flange_a);
  connect(driver.clutch, clutch.f_normalized);
  connect(driver.gasPedal, engine.pedal);
  connect(simplifiedwheel_road.flange_a1, chassis.flange_a);
  connect(roadGrade.grade, chassis.grade);
  connect(driver.actualSpeed, chassis.speed);
  connect(driver.brakePedal, brake.f_normalized);
  connect(brake.flange_a, clutch.flange_b);
  connect(brake.flange_b, simplifiedwheel_road.flange_a);
end vehicle1;

within Powertrain;
model vehicle2
  Powertrain.Driver.RoadGrade roadGrade;
  Powertrain.Driver.Driver driver;
  Powertrain.Components.Engine engine;
  Modelica.Mechanics.Rotational.Components.Clutch
      clutch(mue_pos=[0, 0.3], peak=1.1, cgeo=0.176, fn_max=6973);
  Powertrain.Components.GearBox gearBox;
  Powertrain.Components.SimplifiedwheelRoad
      simplifiedwheel_road(vehicle_mass=1520, radius=0.3);
  Powertrain.Components.Chassis chassis(vehicleMass=1520);

equation
  connect(engine.flange_a, clutch.flange_a);
  connect(clutch.flange_b, gearBox.flange_a);
  connect(driver.gear, gearBox.gear);
  connect(driver.clutch, clutch.f_normalized);
  connect(driver.gasPedal, engine.pedal);
  connect(gearBox.flange_b, simplifiedwheel_road.flange_a);
  connect(simplifiedwheel_road.flange_a1, chassis.flange_a);
  connect(roadGrade.grade, chassis.grade);
  connect(driver.actualSpeed, chassis.speed);
end vehicle2;

Merge these two modelica models to produce a new Modelica model
named vehicle within Powertrain.
The new Modelica model should represent a drive train.
```

This sometimes produces Figure 19.

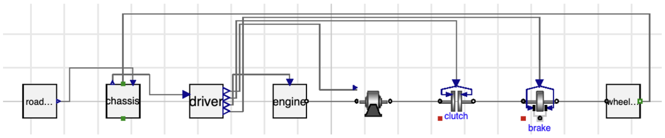


Figure 19: A valid powertrain generated by ChatGPT.

We can extend this qualitative behavior analysis to the other components in the library. For instance, the unit test for the engine is shown in Figure 20 and its behavior is shown in Figure 21.

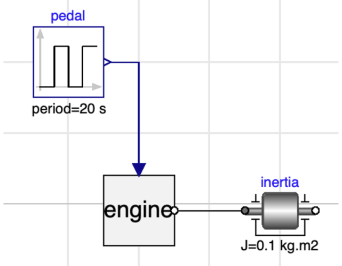


Figure 20: Unit test model for engine

When the engine pedal is zero, the output can be zero or positive and the first derivative can be positive, zero or negative. However when the engine pedal is positive at around 3 seconds, the output is accelerating. This suggests that the engine acts as an accelerator.

The unit test for the clutch is shown in Figure 22 and its behavior is shown in Figure 23.
When the clutch input is zero, then the output is zero. When the clutch input is positive then the output is positive and accelerating. This suggests that the clutch acts as an accelerator or as an on/off switch.

The unit test for the gear box is shown in Figure 24 and its behavior is shown in Figure 25.

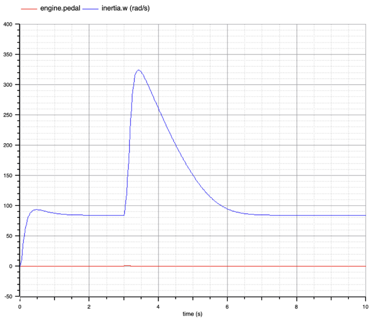


Figure 21: Behavior of engine unit test

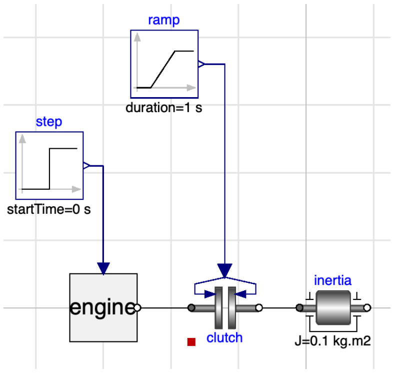


Figure 22: Unit test model for clutch

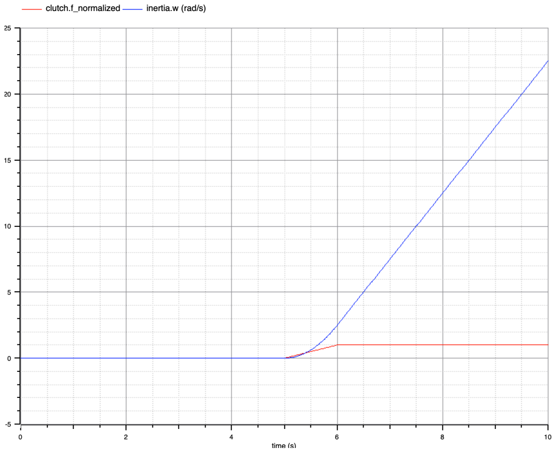


Figure 23: Behavior of clutch unit test

When the gear box input is zero, then the output is zero. When the gear box input is positive, then the output is positive and accelerating. When the gear box input is 2, then the average output is higher than when the gear box input is 1. This suggests that the gear box is acting as some sort of selector.

The qualitative behaviors of the brake, engine, clutch, and gear box can be detected in the desired behavior in Figure 11. This suggests that all of these components are necessary to make a power train. If a candidate design is missing one of these behaviors, then it is worth trying to add the corresponding component.

This sort of qualitative analysis of component behavior can be extended to multi-component behavior as well. It works if a library of known designs has designs with qualitative behavior that can be detected. If a qualitative behavior is in the target behavior but not in a candidate's behavior then we can try adding the known design with that behavior to the candidate design. If a candidate design has a qualitative behavior that the target design does not have and the candidate has a known design with that behavior embedded in it then we can try deleting the known design from the candidate design.

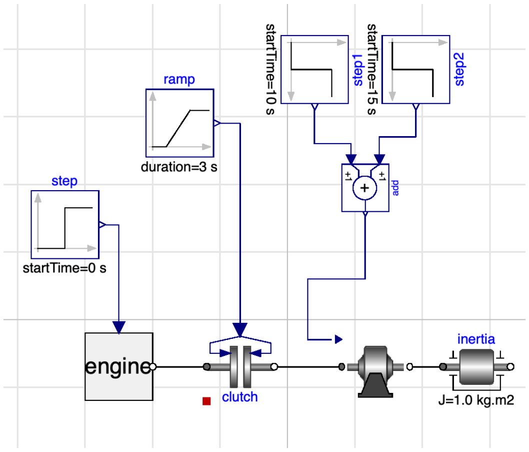


Figure 24: Unit test model for gearbox

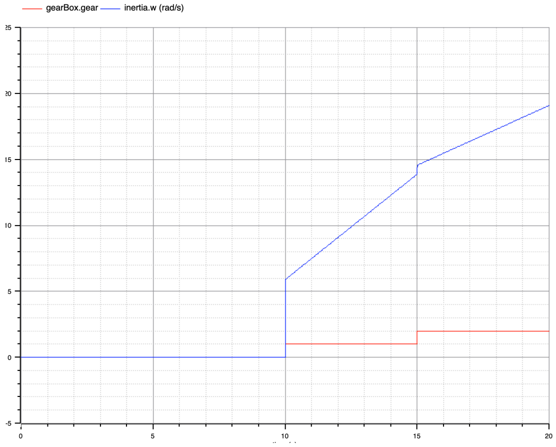


Figure 25: Behavior of gear box unit test

8 Conclusion

The Sizer algorithm can be surprisly expensive and numerical simulation is often fragile within the optimization loop. Therefore, it makes more sense to qualitatively simulate every design before attempting to find the values needed for the parameters. [Klenk *et al.*, 2012].

This paper has illustrated that ChatGPT can be a powerful tool as part of a automated design process. In future work we plan to perform experiments on a wide variety of design problems and determine how large a space of designs can be covered. Introducing more QR promises to greatly speed up the search for possible designs.

References

[de Kleer, 1986] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28(2):127–162, 1986.

[Devlin *et al.*, 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[Fritzson, 2004] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, Piscataway, NJ, 2004.

[Klenk *et al.*, 2012] Matthew Klenk, Johan de Kleer, Daniel G Bobrow, Sungwook Yoon, John Hanley, and Bill Janssen. Guiding and verifying early design using qualitative simulation. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1097–1103. American Society of Mechanical Engineers, 2012.

[Roumeliotis and Tselikas, 2023] Konstantinos I. Roumeliotis and Nikolaos D. Tselikas. Chatgpt and open-ai models: A preliminary review. *Future Internet*, 15(6), 2023.