Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 354

Systems Group, Department of Computer Science, ETH Zurich

Optimizing Declarative Power Sequencing

by

Moritz Knüsel

Supervised by

Daniel Schwyn
Dr. Michael Giardino
Dr. David Cock
Prof. Dr. Timothy Roscoe

March 2021 - September 2021

D INFK

Abstract

Managing a modern computing platform's power and clock infrastructure is an increasingly complex task. Failure to manage it correctly can result in grave consequences, up to the destruction of vital components. The increasing complexity, coupled with a desire for remote monitoring and management, has led to the task of managing power moving away from hard-wired solutions or simple hardware controllers, and on to baseboard management controllers (BMCs). These are fully-fledged computers in their own right, often running commodity software and connected to a network. Given the importance of their tasks, it is desirable that the BMC software be able to provide high assurance that it will operate correctly.

One avenue to improve confidence is to have power sequences generated from a declarative platform model, instead of having humans write them. This thesis aims to improve on previous work in the area of declarative power sequencing by introducing optimization capabilities into the process, allowing us to automatically generate power states that optimize objectives such as power usage.

Acknowledgements

I would like to thank the Systems Group at ETH Zurich for allowing me the opportunity of working on a fascinating topic like this, for the inside look at systems research, and for trusting me to fiddle with the power management on Enzian.

I would also like to give special thanks to my immediate supervisors, Daniel Schwyn and Dr. Michael Giardino, for their continued support and insightful discussions over the course of this work.

Contents

1 Introduction

How is a computer to be turned on? On modern computing platforms, this question is not trivial and requires significant engineering effort. Computing resources such as CPUs or FPGAs require a variety of different supply voltages and clock signals, each of which may be supplied by a different device. All of these devices need to be configured correctly and turned on in the right order. Failure to do so can result in consequences ranging from the system operating less efficiently, to physical destruction of circuits if devices are subjected to voltages they're not rated for.

The task of controlling and monitoring a platform's devices and orchestrating this power sequence is often delegated to a baseboard management controller, or BMC for short. Given the control the BMC has over the whole system, it is desirable to have high assurance that the software on it works correctly and reliably. Specifically, the power sequencing should be as trustworthy as possible. However, coming up with a correct power sequence is most often a largely manual process of reading every relevant datasheet, understanding the interplay between all the devices, designing a power sequence, and writing code to perform the sequence.

This process requires significant effort and it's easy to overlook something. Declarative power sequencing is an avenue to combat these issues. The idea behind declarative power sequencing is to describe a platform's power and clock infrastructure in an abstract model, from which power states and sequences can be automatically derived. This allows us to change the problem of power sequencing. Instead of coming up with a sequence ourselves, we have to write up a description of our platform in an abstract framework, and have a tool derive a sequence from that. Ideally, we would also have formal guarantees that our generated power states and sequences respect the semantics of the abstract model. Declarative power sequencing falls under the more general umbrella of declarative power management. This also encompasses other areas such as generating device drivers for devices like voltage regulators, or automatic failure handling.

Figure 1 shows a diagram that visualizes how this all fits together. In this work, we're looking at the topmost two components, state and sequence generation.

1.1 Contributions

This thesis builds on earlier work by Jasmin Schult [1]. We extend her work with a different model, allowing for optimization objectives to be applied to both power state and sequence generation. We'll make use of this added capability to attempt to optimize a platform's power consumption, as well as to find acceptable ranges for voltages in the system. These can be used to inform monitoring activities, such as checking if a regulator has come up correctly, or to detect faults. We demonstrate that this approach is still performant enough to be used comfortably in an offline setting, and that our optimization objectives can lead to power savings. We also lean more heavily on OMT solvers to do computations for us, in an effort to reduce the amount of code that we have to convince ourselves is working.

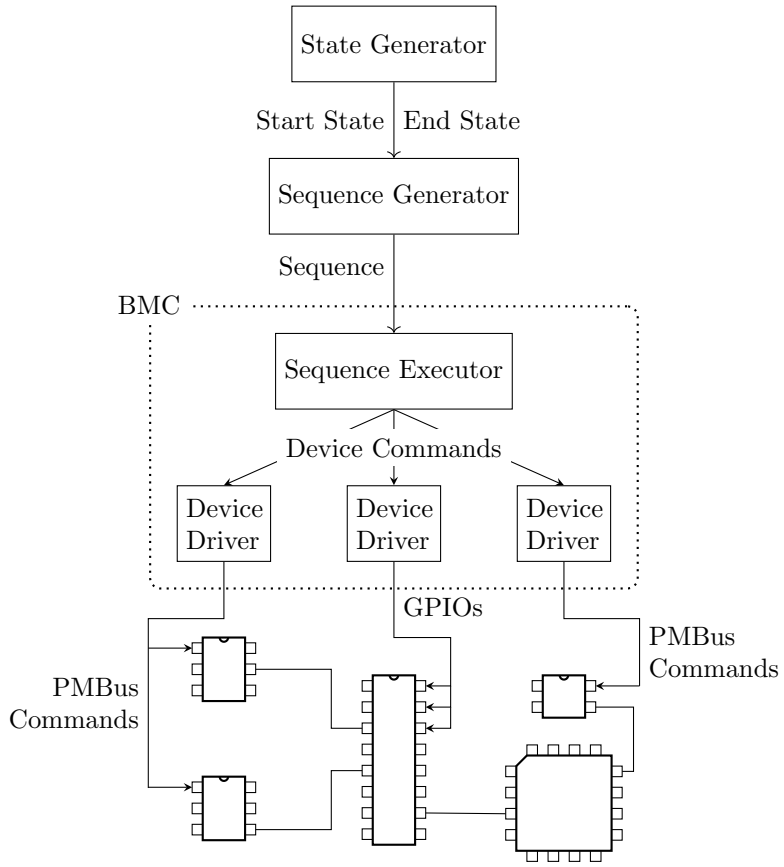


Figure 1: Illustration of how we imagine our work might be used. States and Sequences are generated, and then executed by software running on the BMC.

1.2 Background

1.2.1 Optimization Modulo Theories

Optimization Modulo Theories (OMT) is an extension to Satisfiability Modulo Theories (SMT) [2, 3, 4]. SMT concerns itself with the satisfiability of first order formulas augmented with additional theories. These theories include linear integer arithmetic, bitvector arithmetic, the theory of arrays, and many more. For each theory, specialized methods that are tailored to that theory can be used to solve problems more efficiently than trying to encode the theory in first order predicate calculus.

An SMT instance is a collection of variables and assertions about those variables. An assertion is a term built up from variables and functions. In the core logic, we only get boolean variables and a few boolean functions over them. By relying on additional theories, we gain access to types like arithmetic integers and fixed width bitvectors. The theories also introduce new functions for us to use, like adding and comparing integers.

An SMT solver takes such an instance, and tries to decide if there exists a model for the instance, that is an assignment for every variable that makes every

asserted term true. How quickly this can be decided depends on the size of the problem as well as on the additional theories used. State-of-the-art SMT solvers often have sophisticated heuristics to speed up this process. While it is great to get answers faster, it can make the performance unpredictable, where similar problems can take vastly different amounts of time to solve, if the heuristics happen to favor one of them.

OMT extends SMT by allowing us to solve for models that are optimal with respect to some objective function. We will use the SMT solver Z3 [5], which has been extended with optimization capabilities [3], to solve OMT instances that correspond to our state and sequence generation problems.

The input to the OMT solver is given in a language called SMTLib [6]. While SMTLib is intended to be used by SMT solvers, it is readily extended to support OMT specific tasks such as defining objective functions.

1.2.2 Enzian

Enzian is a research computer in development by the systems group at ETH [7]. It combines a server class CPU with a large FPGA, connected by a high speed interconnect using the native cache coherency protocol of the CPU. Each side is equipped with a lot of RAM and network bandwidth. Additionally, both sides have voltage regulators and clock sources to supply the required voltages and clock signals. These are centrally controlled by the Enzian BMC. At the moment, it runs hand-written power management tools [8].

1.3 Related Work

In the broader area of improving the state of the art in reliable BMCs, there are efforts to provide a formally verified base, by relying on seL4 instead of Linux and formalizing properties of a BMC such as error handling [9, 10].

In modelling a platform's power infrastructure, there is a formal model called p-FSM [11], used to describe components and their interactions. It has been used to formally verify the correctness of a power sequence, but we believe that it may be too flexible to efficiently generate states and sequences.

Then there is the aforementioned work by Jasmin Schult, which has been expanded on in [12]. They solve the state generation problem by establishing constraints on the wires, which are informed by the requirements and affordances of the devices. To generate sequences, every change in a wire is associated with an *Initiate* and *Complete* event. The Initiate event signifies the trigger for a state change, and the Complete event signifies that the state change has happened, and that we confirmed that it happened, for instance by checking the voltage. Devices then impose ordering restrictions on these events, from which a sequence is derived.

They show that their sequence can be computed quickly, and can turn on an Enzian machine successfully. An experience report about the relative difficulty of adapting to a major change in the power management API on the BMC, compared between adapting a handwritten sequence and adapting the abstract model, is included as well.

We expand on that work by introducing a different model to describe devices and platforms. We also add optimization capabilities, enabled by relying on OMT solvers. Using these capabilities, we can introduce optimization objectives

into our state and sequence generation, allowing us for example to search for states that use less power.

Of course, these added capabilities come with a cost, and our implementation does run slower. However, as we will show, it is still reasonably quick, and adequate for offline use. We also demonstrate that we can save power, by evaluating our generated states on an Enzian machine.

In the area of solver-aided program generation, there are a number of frameworks to help in developing program generators, symbolic execution engines, and many more. One such framework is Rosette [13], a framework for writing solver-aided languages implemented in Racket. While frameworks like this can be very useful, we preferred the control over the solver that we gain by interacting with it directly.

2 The Model

Our model is primarily concerned with *devices*, *wires* and *platforms*. A **device** is an object which has some sort of behaviour, and usually has inputs and outputs which shape the behaviour, and allow it to influence other devices. A **wire** connects devices together, and carries information from one device's output to other devices' inputs. A **platform** is a collection of devices and wires that models some physical assembly that we're looking to control. A device in our model doesn't necessarily directly correspond to a specific IC on a PCB. Rather, a device usually abstracts the actual IC together with peripheral ICs and discrete components into a single logical package, hiding the actual implementation details. There are also cases where a regulator with complex behaviour can be described as multiple simpler devices working together. Similarly, our notion of wires can abstract implementation details. For example, a wire that transmits a digital on/off signal can simply be modelled as a logic wire. Since we assume that the platform has already been designed, we'll assume the endpoints of that wire agree on which voltages correspond to which bit, and that this can't be changed by us.

A platform can be in one of many platform states. A platform state is determined by the states of all devices and wires which comprise the platform. To move between platform states, we have platform sequences. We will keep these sequences at a fairly high level, and assume that there is some sort of lower layer that executes these sequences.

2.1 Goals

The following considerations guide the design of the model:

Expressiveness The model should be able to describe a variety of platforms and devices.

Easy to Describe It should not be too tedious to write descriptions of devices and platforms. It's acceptable for complex devices to have complex descriptions, but simple devices should be easy to describe.

Generalized Mechanisms We want to avoid adding special case features to support one specific device. If possible, we want general mechanisms, even if it requires some effort to map some devices onto those.

Abstraction We want to abstract implementation details that have little or no bearing on power states or sequences.

Efficiently Computable It should be possible to compute power states and sequences in reasonable time.

Naturally, in striving for these goals we may have to make certain assumptions about the kind of platforms we aim to model, and make tradeoffs when two goals conflict with each other.

2.2 Device Classes

To model the devices, we describe every type of device as a *device class*. An individual device can then be described as an *instance* of a class.

A device class is described with a name and a number of specifiers for different attributes of the device class. First, a device comes with pins, which define the connections it can have to other devices, as well as internal values. A device also has a set of states in may be in. It can also monitor some of its pins, and raise alerts if the value on one such pin falls outside certain limits. Finally, to make device classes more flexible, we allow instance variables to be used, which can be customized each time the class is instantiated.

We further divide classes into three separate kinds: *Regulators*, *Controllers* and *Consumers*. These distinctions are made to reflect different affordances and requirements found in devices of each of these kinds. We'll start by describing how regulators are modelled, and later explain controllers and consumers by how they're different from regulators.

2.2.1 Regulators

The regulator device classes are intended to model voltage regulators and clocks.

As an example, we'll use the voltage regulator MAX20751 [14] as used on Enzian. As previously discussed, we abstract away peripherals. We're left with a logic supply voltage VDD33, a power supply voltage VDDH, a VR_ON pin to enable the output, and an output voltage V_OUT. The voltage of V_OUT can be configured via PMBus. To power it on, we would first enable the logic supply. After that we would configure the output voltage. At this point, the power supply voltage has to be on as well, but it is not relevant when exactly it got turned on. Finally, we assert VR_ON to make it turn on the output voltage.

The MAX20751 also allows us to measure the voltages on the output and power supply. It can also raise alerts if these voltages deviate from certain ranges.

2.2.2 Pins

Each device class is equipped with inputs, outputs and internal variables. We'll use the shorthand *internals* for internal variables. Inputs, outputs and internals

are collectively referred to as *pins*. Inputs and Outputs may be referred to as IO-pins. Each pin has a name and a type, drawn from a set of signal types:

$$\texttt{signal-types} = \{\texttt{DC}, \texttt{clock}\} \cup \{\texttt{logic}_n \mid n \in \mathbb{N}^+\} \tag{1}$$

For pins that carry control signals, we use a $\texttt{logic}_n$ type. This can model single bit wires, or multiple parallel wires, depending on n. The `clock` type is for pins that carry AC clock signals. For supply voltages we use the type `DC`. The `DC` type is primarily useful for pins where the actual voltage may vary. This gives us some freedom in modelling. Take for instance a power supply unit that outputs a fixed voltage at 12V. Since this voltage will always be 12V and can not be influenced in any way, we could also choose to model it as a logic signal, since the only thing we're interested in is whether or not it is on. However, doing that makes the description less intuitive, since logic wires are typically expected to be low power.

Depending on the type and what kind of pin it is, it is also equipped with additional information.

Inputs that are not of type $\texttt{logic}_n$ specify a minimum and maximum value that the device can accept.

Outputs of type `DC` and `clock` may specify an output accuracy, to indicate how far the voltage or frequency might deviate. Internals always specify an initial value, which specifies the default value of the internal. For each kind of pin we add a specifier describing the pins. The ellipsis indicates that the preceding construct may occur any number of times.

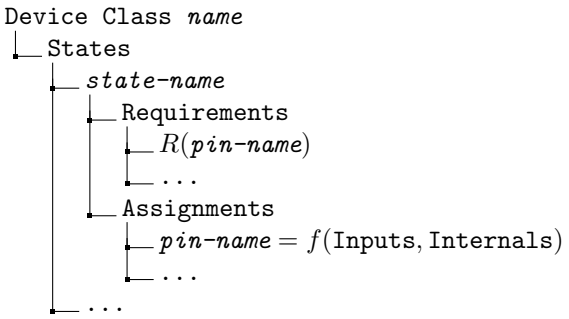
- `Device Class` *name*
  - `Inputs`
    - *name*, *type*, $(min, max)_{opt}$
    - ...
  - `Outputs`
    - *name*, *type*, $accuracy_{opt}$
    - ...
  - `Internals`
    - *name*, *type*, *initial-value*
    - ...

For the MAX20751, the modelling of inputs and outputs is straightforward. Additionally, we use an internal variable VID to model the setting for controlling the output voltage. The input limits and output accuracy can be taken from the datasheet.

- `Regulator MAX20751`
  - `Inputs`
    - VDD33 `DC (0V, 4V)`
    - VDDH `DC (0V, 23V)`
    - VR_ON $\texttt{logic}_1$
  - `Outputs`
    - V_OUT `DC` $\pm 5\texttt{mV}$ `for` VID < 1; $\pm 0.5\%$ `otherwise`
  - `Internals`
    - VID `DC 0.9`

2.2.3 States

A device may be in one of several states, depending on its inputs and internal variables. A state is characterized by a name, a set of requirements on the inputs and internals, and a set of assignments to the outputs. The requirements are given by predicates on individual inputs or internals. Such a predicate R can either require a pin to be a certain value, or require it to fall in a certain range of values. The values assigned to the outputs shall be the result of a linear function over the inputs and internals. We require linearity since non linear functions are harder for OMT solvers to solve.



For regulators, we only allow states to be taken from a fixed set of four states, Off, Powered, Configured and On. Additionally, if a regulator has any internal variables, it must have at least the states Off, Powered, Configured. These states capture a common behaviour of many straight-forward voltage regulators. They start out Off, until the supply voltage is applied. This is when they enter the Powered state. At this point they may be configured, for instance via PMBus, entering the Configured state. Now, asserting an enable signal causes them to transition to the On state, where they start regulating their output.

Transitioning from Powered to Configured is the main way to change a device's internal variables. The other ways are discussed in section 4, when we talk about power sequences.

It's allowed to leave out some of the four states, to accommodate devices that don't need them. For instance, a device might not need any configuration, or it might start regulating as soon as it's configured. Depending on the platform, some devices might be on standby power, so it can make sense to model them without an Off state.

The Configured state is special. Its requirements are the same as the requirements of the Powered state, if that is present. To still make the 'if and only if' relation between states and requirements hold, each regulator with a Configured state needs to have an internal variable configured of type $\texttt{logic}_1$. The Off and Powered states require that configured be 0, and Configured and On require it to be 1. Since these apply to all regulators, we don't require them to be specified explicitly in a device class description.

We also restrict which transitions between these states are allowed. More precisely, we define a set of allowed paths through the states. These are as follows:

Path	Condition
Off → Powered → Configured → On	None
On → Configured → Off	None
On → Powered → Off	Device does not have the Configured state
On → Configured → Powered → Off	Output assignments must be the same in both Configured and Powered

Devices with less than four states use subpaths of these formed by removing the missing states.

We could also include the path On → Off. However, this path can lead to potentially dangerous situations. For instance, a regulator might get turned off by cutting its logic supply, while it is still actively providing power, and the power supply is still up. We decided that we would rather not have that happen.

The main reason for restricting the states of regulators is sequence generation. Thanks to the predefined paths, we don't have to figure out the paths on our own based on the requirements. This not only makes it easier to come up with a sequence, but it also simplifies defining device descriptions. In an earlier iteration of the model, where these constrains were not in place, it was often necessary to engineer the input requirements of a device's states in such a way that the sequence generation traversed the device's states correctly. At times this necessitated overconstraining states, meaning legitimate input configurations were ruled out by the power state search. It also complicated device description in general.

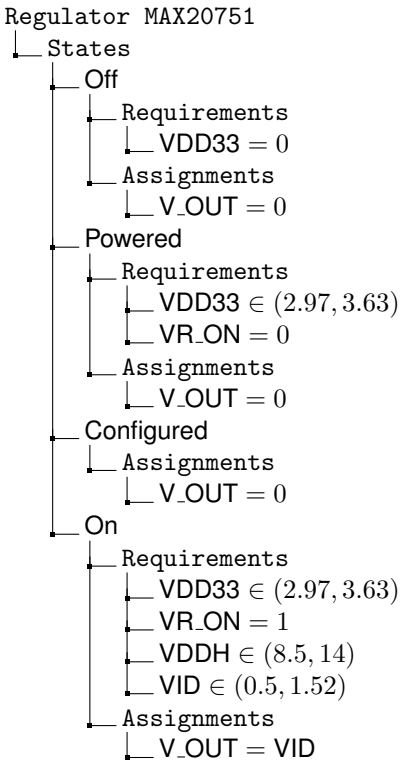
A device occupies exactly one of its states at any given time. Which state this is is determined by the inputs and internals. A device is in state s if and only if the requirements for state s are fulfilled.

This guarantees that we can always determine the state of a device if we know the values of its inputs and internals. It also tells us what requirements we need to fulfill if we want a device to be in a specific state. Which state a device is in will also direct the values of the device's outputs.

We don't impose any constraints from the outputs back on the state, since many devices have multiple states with the same output assignments. Some don't even have outputs.

For sequencing, a device enters a state as soon as the requirements for that state are met. At this point the outputs assume the values given by that state's assignments.

The MAX20751 fits neatly into our four state model. It starts off in the Off state, until the logic supply is applied, which transitions it into the Powered state. For the Off state, we only require that the logic supply is off. Without the logic supply, the values of the other inputs is irrelevant. For the Powered state, we require that VR_ON is turned off. Here the VID value can be set. Once the power supply is enabled, we can turn on VR_ON to transition to the On state. The On state also requires VID to be set to an appropriate level. This is what the description of the MAX20751's states looks like:



2.2.4 Monitors and Alerts

Another capability of many devices is monitoring. Devices can take measurements of voltages or currents, and either report them on request, or raise alerts when they fall outside defined boundaries. Additionally, voltage regulators often have so-called power-good pins that are asserted when the device's output voltage is ready.

We want to model these features for two reasons. First, the sequences we generate should contain instructions that indicate that we should wait for a certain voltage to change. For this we need to know which voltages are monitored by which devices. Second, our model gives us enough information to compute boundary levels for devices that can raise alerts. Therefore it makes sense to compute those here, rather than pushing it down to a lower layer.

We'll say that a device *monitors* something if we can query the device for a measurement. The term *alert* will be used when a device can raise alerts independently.

A device may monitor any of its IO-pins. It can measure different qualities, like voltage or current. We'll call the capability of a certain device to measure a certain quality on a certain pin a *monitor*. Each monitor has a monitor-type, which specifies which quantity it measures. The possible monitor-type depends on the type of the pin being measured. For pins of type `DC`, we may measure `voltage` or `current`. For pins of type $\texttt{logic}_n$, we can use the monitor type `value`. For `clock` type pins we may measure `frequency`. Finally there is `binary`, which may be used with any pin. The `binary` type is special, and indicates that the monitor will only be able to indicate whether or not the

measured pin has reached its target value. This is used to model things like power-good pins.

For alerts, a device may raise alerts for IO-pins of type `DC`, but only for voltage. The reason for this restriction is that the model can only provide us with bounds for voltages, but not for currents. This is because the model as it is now does not model currents, mainly because they're largely outside our control.

It may also raise alerts for `clock` pins, if the frequency falls out of a defined range.

We extend the device class description with monitor and alert clauses:

```
Device Class name
├─ Monitors
│  └─ monitor-type, IO-name
└─ Alerts
   └─ IO-name
```

Recall that the MAX20751 can both monitor the voltages on VDDH and V_OUT, as well as raise alerts if the voltage fall outside configurable limits. We describe these capabilities with the appropriate specifiers:

```
Regulator MAX20751
├─ Monitors
│  ├─ voltage VDDH
│  └─ voltage V_OUT
└─ Alerts
   ├─ VDDH
   └─ V_OUT
```

2.2.5 Instance Variables

Additionally, we need a mechanism to customize device instantiations. In the case of the MAX20751, our description is a bit limiting. In reality, the initial value of the internal variable VID is dependant on an external register, which may be different from chip to chip. For this reason we include instance variables, which can be assigned when a device class is instantiated.

```
Regulator MAX20751
├─ Internal Variables
│  └─ default
└─ Internals
   └─ VID DC default
```

With this we have all we need to describe regulators. Most of what we've seen will also be used to describe controllers and consumers. The generic shape of device classes can be seen in figure 2. It shows the basic template which is common to all three kinds of device classes, and which we will take as a base for controllers and consumers.

To illustrate the abstraction that has taken place, compare a visualization of our description to a schematic from Enzian in figure 3. Compared to the complex assembly of ICs and discrete components, our high level view captures what is essential for our purposes.

Refer to figure 4 for the full description of the MAX20751.

- Device Class *name*
  - Internal Variables
    - *name*
    - ...
  - Inputs
    - *name*, *type*, $(min, max)_{opt}$
    - ...
  - Outputs
    - *name*, *type*, $accuracy_{opt}$
    - ...
  - Internals
    - *name*, *type*, *initial-value*
    - ...
  - States
    - *state-name*
      - Requirements
        - R(*pin-name*)
        - ...
      - Assignments
        - *pin-name* $= f($Inputs, Internals$)$
        - ...
    - ...
  - Monitors
    - *monitor-type*, *IO-name*
  - Alerts
    - *IO-name*

Figure 2: Structure of device classes.

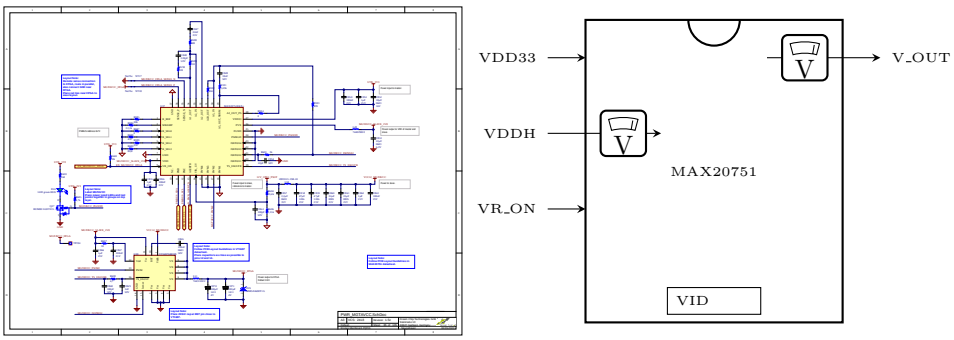


Figure 3: Compare a visualization of the device description of the MAX20751 as used in Enzian (on the right), to a MAX20751 from the Enzian schematics. All the details like analog components and slave ICs have been abstracted away.

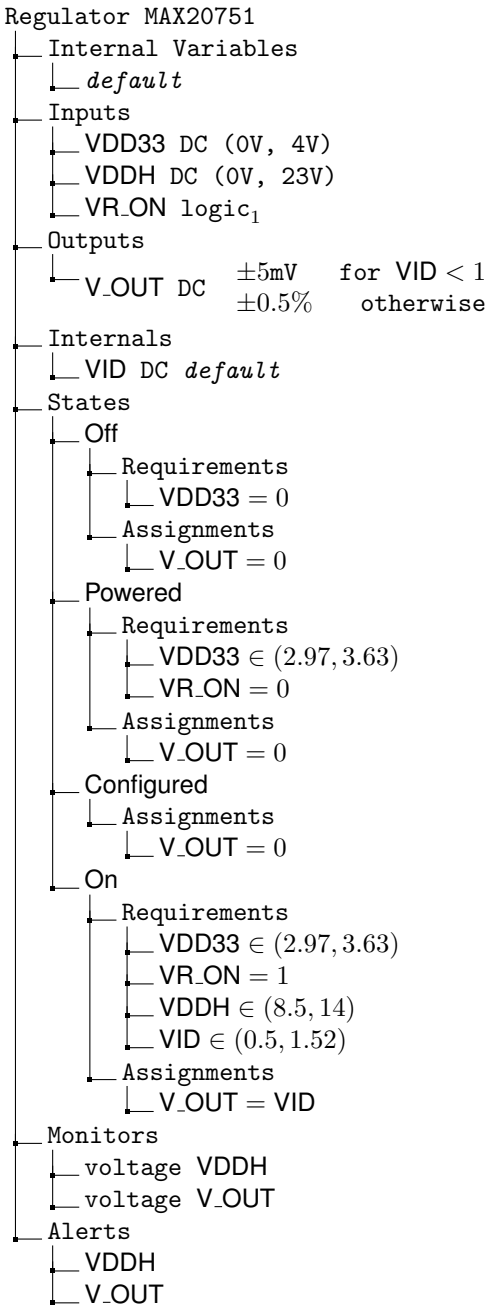


Figure 4: A sample description of the MAX20751 regulator.

2.2.6 Controllers

There are some devices that don't fit within the constraints imposed by the regulator descriptions. Take for instance the Enzian BMC itself. It controls a number of logic signals, every one of which might be enabled or not independent of the others. We might solve this by having the value of each logic signal be controlled by an internal variable. However, this still requires that all the logic signals change state at the same time. For devices like the BMC, we really want to be able to control the outputs at any time, without having to change the device's state.

To model such devices, we use the concept of a *controller*. For controllers we use the same restricted state set as for regulators. Unlike regulators, we drop the output assignments for controllers. Instead, we assume that the outputs are zero in the Off and Powered states. In the other states, the outputs may assume any value and can be changed at any time.

Changing the value of a controller's output is also one of the main ways to interact with the platform, meaning it's one of the actions we might take in a power sequence. This reflects how the Enzian BMC controls the platform. It can change its own GPIO pins, which are modelled as outputs of the BMC, or it can send messages to the ispPACs [15], telling them to change their outputs.

An alternative solution would be to control each output with an internal variable, and allow internal variables to be changed at any time, for any device. This is, in a sense, closer to reality, as the internal devices of regulators could in fact be changed at any time, as long as the regulator has power. However, this makes the generation of power sequences more complicated. For controllers, we don't gain anything from this alternative approach, it only makes the description more verbose due to the additional internal variables. For regulators, it would allow for sequences that turn the output on first, and only set the desired voltage after. It's not clear if this would be useful, and it forces us to consider the intermediate value, which might violate the target device's input requirements, or trip alerts on other devices connected to the output.

In essence, the difference in how flexible controllers are in their outputs as compared to regulators, warrants this split.

2.2.7 Consumers

The last kind of device class is the *consumer*. We use this category to describe devices that are at the leaves of the power tree, consuming the voltages generated by the regulators. These devices typically come with complex sequencing requirements.

We'll use a simple, made-up CPU as an example. The CPU shall have two supply voltages: VDD_CORE at 0.9V and VDD_IO at 3.3V. It also has a clock input CLK at 100MHz, as well as two control signals. DC_OK to indicate that the supply voltages are ready, and RESET_N to actually start the CPU. To turn it on, CLK, VDD_CORE, VDD_IO, DC_OK and RESET_N have to be enabled in order. To turn it off, first RESET_N has to be turned off, followed by DC_OK. Then the two supply voltages can be turned off in any order, followed by CLK. Additionally, we'd like to be able to park the CPU in a reset state. This will necessitate a separate state just before it is fully turned on, so we can generate platform states in which the CPU is in the reset state.

This device doesn't fit well into the simple four state model we use for regulators or controllers. Just to turn it on we need to change 5 pins in a precise order, so modelling it as a single four state device won't suffice. While it may be possible to split the CPU up into multiple devices connected by control wires, it would make the resulting description much more complicated, especially once the number of inputs and complexity of the sequencing requirements grow.

Therefore we introduce a third device class tailored to consumer devices. For consumers we allow any states, meaning the restrictions on states imposed on regulators and controller don't apply. To control transitions between the states of a controller, allowed transitions are indicated by sequence specifiers. These sequence specifiers also describe the order in which inputs are switched on or off.

A sequence specifier consists of a starting state and an end state, plus a list. An element of this list may either be a set of inputs, or a wait instruction, signifying that there is a minimum time that has to pass between the surrounding steps.

```
Sequence from-state → to-state
├─ (name, ...)  | Wait time
├─ ...
```

To transition the device from the starting state to the end state, the specified inputs must be switched in the order they appear in the list, and the timings specified by the Wait instructions have to be observed. Inputs within the same set may be switched in any order relative to each other.

See figure 5 to see how our example CPU can be described using these mechanisms.

2.3 Platforms

A platform is comprised of devices and the wires that connect them. The devices are instances of device classes. They are specified with an instance specifier:

```
Instance device-class instance-name
├─ instance-variable = value
├─ ...
```

An instance specifier specifies a device with a given name, of a given device class. It also assigns all the instance variables of the device class. Devices are connected by wires. A wire specifier contains a name for the wire, as well as a single starting point, specified by a device name and output pin, and a number of end points, each specified by a device and input pin. Wires are also typed, using the same set of types as we use for pins. The type doesn't have to be specified, it's inferred from the type of the starting point. The types of the start and end points all have to match.

```
Wire name
├─ From device pin
├─ To device pin
├─ ...
```

A wire takes the value assigned to its starting pin, and imposes that value on the inputs at the end points.

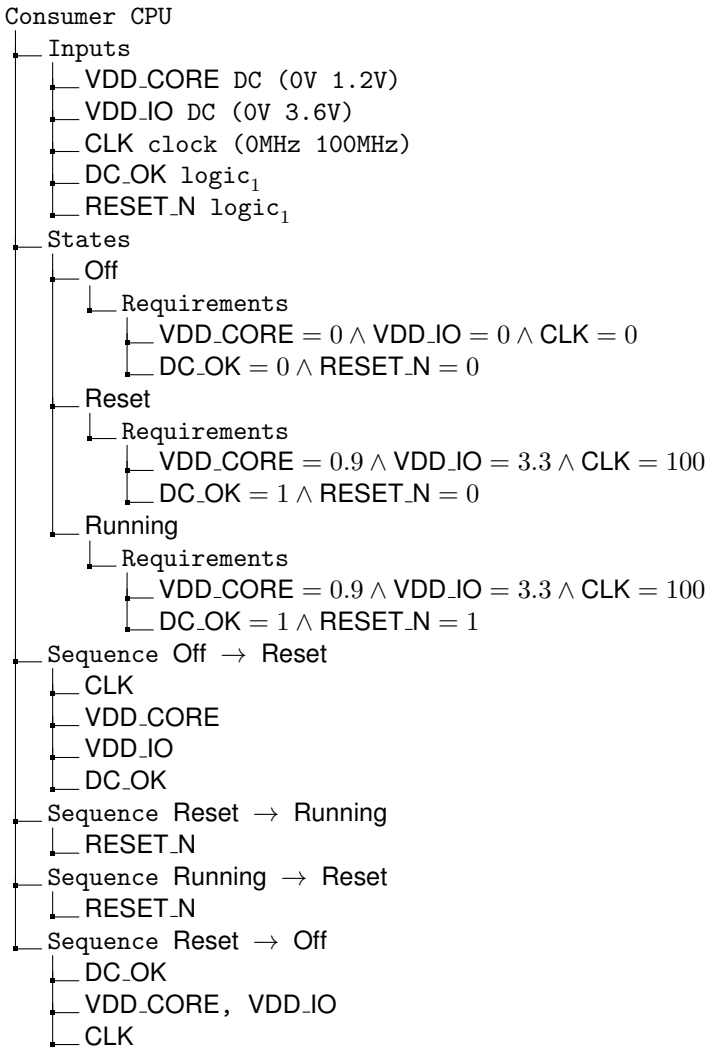


Figure 5: A description of a CPU as an example of a consumer.

2.4 Platform States

A platform state describes the full concrete state of a platform. It assigns a value to every wire, and fixes the states and internal values of every device. These states are generated from the rest of the model. In order to direct the state generation procedure to come up with states we're interested in, we add platform state specifiers to the platform description. These contain constraints that can fix the state of a certain device, or fix the value of a certain wire. Usually we want to constrain the states of the consumers, and let the state generation figure out what state the other devices should be in. This may not uniquely determine the states of all other devices though. Take for instance a controller who's only task is to let us monitor a certain wire. If that wire has other monitors, it's not clear whether that controller should be powered on or not. So we can constrain its state to determine that.

```
Platform State name
├─ device-name in state-name
├─ ...
├─ wire-name = value
└─ ...
```

In addition to the platform states given by the platform state specifiers, we also define the so called Initial state. This is the state in which all devices are in the Off state, and all internal values are assigned their specified initial value. It is supposed to describe the platform when it's powered off.

2.5 Sequences

A sequence describes the actions that need to be taken to transition the platform from one state to another. As shown in figure 1, we assume that this sequence will be executed by some lower layer that has more specific information as to how to run the actions, so the actions are fairly high level.

There are four actions. `Set`, `Configure`, `Monitor` and `Wait`.

`Set` *controller, output = value*

A `Set` action sets the output of a given controller. How this happens is left to a lower layer to decide, which might for example send a PMBus command or set a GPIO pin.

```
Configure device-name
├─ Set internal = value
├─ ...
├─ Limits IO-pin (min, max)
└─ ...
```

A `Configure` action instructs the lower layer to configure a regulator or controller by setting its internal variables, as well as setting the limits on its alerts. The limits are determined by the input limits on the devices connected to the watched pin, as well as the output accuracy of the output feeding the wire.

```
Monitor wire-name
├─ device-name pin type (min, max)
├─ ...
├─ device-name pin type value
└─ ...
```

The `Monitor` action tells the lower layer to wait for a certain wire to reach a certain value, or a range of values. It also gives a list of devices that can monitor the wire at this point, and how they can monitor it. If a device doesn't have a Configured state, we consider it to always be able to monitor. Devices that do have one have to be in either the Configured or On state in order to be able to monitor.

`Wait` *time*

A `Wait` action instructs the lower layer to wait for the specified amount of time. This may be necessary if a consumer specifies a wait step in one of its sequences.

A sequence consists of a list of pairs, where each pair is made up of an action and a priority number. The actions are to be executed in order of their priority numbers. Actions with the same priority may be executed in any order relative to each other, or even at the same time. This potentially allows for lower level optimizations, for instance two equal-priority `Set` actions to the same controller could be combined into a single command that sets both outputs, if the controller supports that.

In order to describe the sequences we want to search for, we use sequence specifiers. A Sequence specifier gives that start and end state of the platform by name. The names refer to the platform states given by the platform state specifiers, or the initial state. Additionally, a sequence specifiers contains optimization goals. We can specify that a certain device should reach a certain state as soon as possible. Usually we'd want to get the relevant consumers turned on as soon as possible.

We may also describe additional constraints over the whole platform. We can specify that a certain set of events should happen before another set of events. The events are devices entering certain states. This is useful to express constraints that the model has no other ways of knowing, or that would require undue cleverness to encode with other means. For instance, there are some I^2C devices that will render the bus unusable while they're powered off. We can ensure that we never try to use the bus while the problematic devices are powered off, by specifying an ordering constraint that says that all the devices on the bus may only enter their Configured state after all problematic devices have reached their Powered state.

```
Platform Sequence name from-state → to-state
├─ Minimize device state
├─ ...
├─ Constraint name
│  └─ (device, state) ... < (device, state) ...
└─ ...
```

3 Generating Platform States

The problem of generating a platform state consists of taking a platform state specifier, as shown in section 2.4, and coming up with an assignment that assigns a value to every wire and pin on the platform, and a state to every device. The assignment must respect the input limits of every input pin, and the requirements of each device's state. It also must follow the constraints laid out in the platform state specifier. Additionally, we want to find the maximum and minimum value for every wire that has a monitor or alert.

Generating a full platform state is a two step process. First we construct an OMT problem to find an assignment for wires, pins and states. We then use this to construct a second problem to find minima and maxima for monitors and alerts.

The reason for doing this in two steps is that, while it is possible to combine them into a single OMT instance, doing so makes it much slower to solve. By splitting it into two steps, we can simplify the limit search by providing the solver with a more constrained search space.

We'll now look at how we construct the first OMT instance.

We represent the value of wires and pins according to their type. A voltage on a wire of type `DC` is represented by an integer, in millivolts. The value of a `clock` wire is represented by an integer in Megahertz. Finally, a value of type $\texttt{logic}_w$ is represented by a bitvector of width w.

Before we start declaring our variables, we define a few meta functions and accessors to be used in the description of the variables and constraints.

Function Name	Description
$\textbf{type}(x)$	The type of a wire or pin.
$\textbf{start}(w)$	The device and pin where the wire w starts.
$\textbf{ends}(w)$	A list of device/pin pairs where the wire w ends.
$\textbf{inputs}(d)$	A list of inputs of device d.
$\textbf{outputs}(d)$	A list of outputs of device d.
$\textbf{internals}(d)$	A list of internals of device d.

Now we define the variables and datatypes we use. For each instantiated device d, we generate a datatype for its states, where the states of d are named $s_{0,d}$ through $s_{n,d}$

$$\texttt{enumeration}\ \mathsf{State}_d : s_{0,d}, ..., s_{n,d}$$

The decision to use an enumeration here does come with drawbacks, because custom datatypes like this are a relatively recent addition to SMTLib. This means that we might not be able to use some solvers, if they don't support the required SMTLib version. An alternative solution would be to use integers and constrain the set of allowed values manually. However, using symbols instead of numbers makes it easier to read the generated OMT instance, as well as the model found by the solver.

Then, for each instantiated device d, we define a variable to represent its current state:

$$\texttt{var}\ \ \mathit{current\text{-}state}_d : \mathsf{State}_d$$

Next, we define a variable of the appropriate type for each input, output and internal.

$$\texttt{var}\ \ \mathit{input}_{d,i} : \textbf{type}(i)$$
$$\texttt{var}\ \ \mathit{output}_{d,i} : \textbf{type}(i)$$
$$\texttt{var}\ \ \mathit{internal}_{d,i} : \textbf{type}(i)$$

For regulators and controllers, we also define a function to assign a numerical value to the device's state. This is useful when encoding optimization goals.

$$\texttt{fun}\ \ \mathit{state\text{-}value}_d(s) : \mathsf{State}_d \to \mathit{Int} = \begin{cases} 0 & \text{if } s = \mathsf{Off}_d \\ 1 & \text{if } s = \mathsf{Configured}_d \\ 2 & \text{if } s = \mathsf{Powered}_d \\ 3 & \text{if } s = \mathsf{On}_d \end{cases}$$

For the wires, each wire w generates a variable reflecting its value.

$$\texttt{var}\ \ \mathit{wire}_w : \mathit{type}_w$$

Now we need to encode the constraints. We start with the wires. For each wire w, we assert that the value at the start point and the values at the end points are all equal to the value of the wire itself.

$$\mathit{wire}_w = \mathit{output}_{\mathbf{start}(w)} \wedge \forall (d, i) \in \mathbf{ends}(w) : \mathit{wire}_w = \mathit{input}_{d,i}$$

For the devices, we start with specifying the absolute limits on the inputs. For each device d, we require that the value of all inputs is within the specified limits, taking into consideration the accuracy of the output feeding the wire connected to the input. We introduce a few additional meta functions to aid in this constraint:

Function Name	Description
input-limits(d)	A list of pairs, consisting of an input and the associated absolute limits, for a device d.
associated-wire(d, i)	The wire connected to an input i in device d.
accuracy(d, o)	The accuracy of an output o in device d, given as a pair of formulas of one argument.
output-high(d, o)	A list of states where the output o of device d is not zero.

Here we have a meta function that returns formulas. When we say that a function returns a formula, we mean that it returns an OMT expression. For instance, for an accuracy of $\pm$ 5 mV, **accuracy** would return $(-5, 5)$. These formulas may take arguments, which are inserted into the formula. For instance, for an accuracy of $\pm 5\%$, we would get $\lambda x. \frac{5x}{100}$. Note that the substitution in formulas happens as we build the OMT instance. Formulas may also contain free variables, referring to other OMT variables we defined.

To encode the input boundaries, we first get the input we want to constrain, together with the lower and upper bound that are specified in the device description. We add an offset to those limits, which is given by the output accuracy of the output that drives the wire connected to the input. Since this offset might depend on the actual value on the wire, we apply the accuracy formulas to that value. With this, we assert the input boundaries as follows:

$$\forall (i, (l, h)) \in \mathbf{input\text{-}limits}(d) : \mathit{input}_{d,i} \geq l + f \cdot a^- \wedge \mathit{input}_{d,i} < h - f \cdot a^+$$

where:

$$\begin{aligned}
f &= \text{an additional safety factor} \\
a^- &= \begin{cases} |\mathbf{F}^-(\mathit{wire}_w)| & \text{if } \mathit{state}(d') \in \mathbf{output\text{-}high}(d', o') \\ 0 & \text{otherwise} \end{cases} \\
a^+ &= \begin{cases} |\mathbf{F}^+(\mathit{wire}_w)| & \text{if } \mathit{state}(d') \in \mathbf{output\text{-}high}(d', o') \\ 0 & \text{otherwise} \end{cases} \\
w &= \mathbf{associated\text{-}wire}(d, i) \\
(d', o') &= \mathbf{start}(w) \\
(\mathbf{F}^-, \mathbf{F}^+) &= \mathbf{accuracy}(d', o')
\end{aligned}$$

Note that a^- and a^+ are not literal values, but also formulas that refer to other variables.

We only apply the accuracy if the pins is actively driven by another device, since it doesn't make sense to take the accuracy into account if the regulator isn't running and regulating the wire.

Since we'll need to encode limits again, we define the function:

Function Name	Description
accuracy-adj(d, i)	Returns formulas a^- and a^+ given a device d and an input i

Next we establish the requirements on states, and how the states assign the outputs. We again introduce a two accessors to get the specified requirements:

Function Name	Description
required-ranges(d, s)	A list of pairs, each consisting of an input and a range of values representing the input requirements of device d in state s. Only returns the inputs where a range is required.
required-values(d, s)	A list of input/value pairs, representing the inputs where device d in state s requires a certain value.
assignments(d, s)	A list of output/formula pairs representing the output assignments of a device d in state s.

For each device d, and each of its states s, we assert that:

$$\begin{aligned} \textit{current-state}_d = s \iff \forall (i, (l, h)) \in \textbf{required-ranges}(d, s) : \\ \textit{input}_{d,i} \geq l + f \cdot a^- \wedge \textit{input}_{d,i} < h - f \cdot a^+ \\ \textit{current-state}_d = s \iff \forall (i, v) \in \textbf{required-values}(d, s) : \textit{input}_{d,i} = v \end{aligned}$$

where:

$$\begin{aligned} f &= \text{ an additional safety factor} \\ (a^-, a^+) &= \textbf{accuracy-adj}(d, i) \end{aligned}$$

We relate state and requirements by $\iff$ to make sure that whenever a device is in some state, that state's requirements hold. Also, whenever a state's requirements hold, the device is actually in that state. This takes care of the requirements. To constrain the outputs, we assert that:

$$\textit{current-state}_d = s \implies \forall (o, \mathbf{F}) \in \textbf{assignments}(d, s) : \textit{output}_{d,o} = \mathbf{F}$$

This is all it takes to encode the platform semantics. Any model satisfying this instance will respect the absolute input limits on all devices. It will also respect the state requirements of each device for the state it is in. To further constrain the instance to comply with a platform specifier, we assert that each specified device is in the state given by the platform state specifier, and every specified wire has the value assigned to it in the specifier.

To find the initial state we assert that each internal variable has the value given as the initial value in the corresponding specifier.

3.1 Optimization Goals

Once we have asserted that the state we generate obeys the constraints imposed by our model, we can formulate optimization goals for our state. First we will define a few goals that aim to guide the solver towards more minimal platform states if something is not sufficiently constrained. How minimal a platform state is roughly correlates to how many steps are required to reach it from the initial state. Ambiguities that affect this arise in two situations. First, a regulator's output may be constrained to be 0V. This can have little to no bearing on the possible states that the regulator might be in. Take an instance of the MAX20751 from section 2.2.1 as an example: It only assigns a non-zero value to its output in the On state. In a platform state where its output is zero, the solver might decide that the device should be in the Configured state, when it could just as well be in the Powered state. If we wanted to reach this platform state from the initial state, we'd incur an unnecessary `Configure` action in the sequence.

To remedy this problem, we pose the following goal for each device d:

$$\texttt{minimize}\ \mathit{state\text{-}value}_d(\mathit{current\text{-}state}_d)$$

Second, a controller could have outputs that are not uniquely determined. This could have two reasons. Either the device at the other end of the output doesn't impose any constraints in the particular state it is in, or the output may not be connected at all. This may be because we wrote its description with respect to the datasheet, but some of the pins are not used on the particular platform the device is used in, or we might have different instances of the same controller class, each of which uses a different subset of the outputs. This would result in a sequence with unnecessary `Set` actions, if the solver decides the unused outputs should be different in the start and end platform states. To remedy this, we minimize each output o of each controller d:

$$\texttt{minimize}\ \mathit{output}_{d,o}$$

Finally, we want to formulate goals to optimize the power usage of the platform. As a first approximation, we minimize every voltage. Thus, for each wire w of type `DC`:

$$\texttt{minimize}\ \mathit{wire}_w$$

To recover the platform state from the OMT model, we extract the value of every pin and wire, as well as the state of every device.

3.2 Limit Generation

We now have computed a platform state that is almost ready to be used. To help in the later sequence generation step, we would like to compute the bounds on alerts and monitors. For this, we pose a new OMT instance. We use the same variables and constraints as before, without the optimization goals. Instead, we

assert the values that we obtained for the platform state we already have. Now, to figure out the limits to be used for alerts and monitors, we need to find the minimum and maximum value of every wire that has at least one monitor or alert. To do this we add two new variables per such wire, to which we will apply the input constraints of the devices connected to the wire. In the end, we can minimize and maximize the variables to find the minimum and maximum allowable value for that wire. For every such wire w, we add two shadow wires:

$$\texttt{var } \mathit{shadow}_w^+ : \textbf{type}(w)$$
$$\texttt{var } \mathit{shadow}_w^- : \textbf{type}(w)$$

Next we need to constrain the shadow wires. For each device d, and each of its states s, we assert that:

$$\begin{aligned}\mathit{current\text{-}state}_d = s \implies \forall (i,(l,h)) \in \textbf{required-ranges}(d,s) : \\ \mathit{shadow}_w^+ \geq l \wedge \mathit{shadow}_w^+ < h \\ \wedge\, \mathit{shadow}_w^- \geq l \wedge \mathit{shadow}_w^- < h \\ \mathit{current\text{-}state}_d = s \implies \forall (i,v) \in \textbf{required-values}(d,s) : \\ \mathit{shadow}_w^+ = v \wedge \mathit{shadow}_w^- = v\end{aligned}$$

where $w = \textbf{associated-wire}(d,i)$. We assert similar constraint for the absolute input limits:

$$\begin{aligned}\forall (i,(l,h)) \in \textbf{input-limits}(d) : &\mathit{shadow}_w^+ \geq l \wedge \mathit{shadow}_w^+ < h \\ &\wedge \mathit{shadow}_w^- \geq l \wedge \mathit{shadow}_w^- < h\end{aligned}$$

where again $w = \textbf{associated-wire}(d,i)$.

Note that we have not accounted for the output accuracy in constraining these shadow wires. The reason is that before, we were looking for the value that a regulator should aim towards. There, we have to account for the accuracy, since the value we set may not be the one that's actually on the wire. In the case of monitors and alerts, we are concerned with the value that is actually on the wire. In fact, if we were to take the output accuracy into account here, we'd end up with a platform where every voltage is driven right down to the alert limit, and the slightest downward deviation would trigger an alert. This would make for a rather unstable system, and is to be avoided.

With these constraints, we can construct optimization goals to figure out the limits. For each wire w with shadow wires, we pose:

$$\texttt{maximize } \mathit{shadow}_w^+$$
$$\texttt{minimize } \mathit{shadow}_w^-$$

Once it is solved, we extract the minimum and maximum values and add them to the rest of the concrete platform state.

4 Generating Sequences

A sequence describes how to transition the platform from one state to another, while satisfying the sequence requirements of every consumer, the input limits

of every device, and the constraints laid out in the corresponding sequence specifier. Additionally, every device except consumers should always be in one of its named states. We allow consumers to inhabit intermediate states while they move from one named state to another. First, we need to make a few assumptions about the platform we're trying to find sequences for. If these assumptions are not met, we may not be able to generate a sequence, or we may generate sequences that violate a device's input limits.

The first assumption is that any wire and pin changes its value at most once over the whole sequence. This simplifies the problem because we don't have to synthesize new values, and make sure these new values conform to all requirements. If the value of a particular pin or wire can only change once, we know that, at any time, the value will either be the starting value of the pin or wire, or the goal value.

Additionally, we assume that the state requirements on a pin also only change once during the sequence. In fact, we require that not only do they not vary semantically, but syntactically. Together with the first assumption, this simplifies our task significantly. For any pin p on a device d, we can conclude that the point where the requirements change is also the point where the the value of p changes. Otherwise, the requirements would be violated. Furthermore, we can find that point by examining the state requirements purely on a syntactic basis, which means we don't have to spend time checking the equivalence of arithmetic ranges, but can just compare them syntactically.

We don't believe these assumptions are too constraining. If a sequence is desired that doesn't fit the assumptions, it may still be possible to split it into multiple sequences with intermediate platform states, where each subsequence satisfies the assumptions.

4.1 Start and Goal States

The OMT instance we generate heavily depends on the platform start and end states of the desired sequence. To facilitate the generation of the OMT instance, we'll use the meta-functions **begin** and **goal**. They shall, given any variable used in the state generation, return the value that variable has in the platform start state, and in the end state respectively. So for instance $\mathbf{begin}(input_{d,i})$ would return the value assigned to input i of device d in the start state, and $\mathbf{goal}(wire_w)$ would give us the value of wire w in the end state.

4.2 Overview

Based on the start and goal state, we determine every event that can happen in the sequence. An event may, for instance, be a wire or IO-pin changing its value, or a device entering a new state. These events are the related to each other. An event can be constrained to happen before, after, or at the same time as another event. How the events are related to each other is determined by each device. We look at the ways a device can reach its goal state from its starting state. Each event is represented by the time when it happens. The solver will assign a time to each event, from which we will recover the sequence.

4.3 Variables

First we define the variables used in the OMT instance. We start with the wires. For every wire w, where $\textbf{begin}(wire_w) \neq \textbf{goal}(wire_w)$, we declare:

$$\texttt{var}\ \ edge_w : Int$$

This variable signifies the point in the sequence where the value of the wire w changes. This is all we need for the wires. We define similar variables for every input and output p of every device d, again only for those that have different values in the start and end state:

$$\texttt{var}\ \ edge_{d,p} : Int$$

Note that no such edge variables were declared for device internals. That's because we model internals as being set all in one go using a `Configure` action. For this purpose, we declare the following variables:

$$\texttt{var}\ \ configure_d : Int$$

We introduce this variable for every device d that will require configuration. We determine whether or not a device will require configuration depending on what kind of device it is, whether it has a Configured state, and its concrete device states in the start and end platform states. We require a configuration step when a device could change from Powered to Configured during the sequence. We may also require this step if the device doesn't change its state at all. Namely, if some of the device's internal variables change, or if any alert limits on the wires for which the device raises alerts change.

Next we define variables to indicate when a device enters one of its states. For every device d, and every state s it could be in, we declare:

$$\texttt{var}\ \ enter_{d,s} : Int$$

This variable signifies the point where device d enters state s. In other words, the point where the requirements of state s are first fulfilled.

We also need one path selector variable per device d:

$$\texttt{var}\ \ path\text{-}select_d : Int$$

This variable is required because a device may have multiple different ways to get from its starting state to its goal state. To take this into account, we'll enumerate these different paths and condition the constraints for each path on the value of the path selector. For every device d that has monitors, we also declare the following variable, for every monitored pin p, and monitor type t:

$$\texttt{var}\ \ monitor\text{-}ready_{d,p,t} : Int$$

This variable will indicate the time where the corresponding pin can be monitored. We'll use this to ensure that we always have a monitor available when we want to change a wire, so we can always confirm the change happened as planned.

Finally, we require some special variables for consumer devices. In consumers, we allow sequences to be specified. To make sure these are respected

by the final sequence, we proceed as follows. For every consumer d, and every sequence specifier S, we declare:

$$\begin{aligned}&\texttt{var } \mathit{front}_{S,0} : \mathit{Int}\\&\texttt{var } \mathit{front}_{S,1} : \mathit{Int}\\&\quad ...\\&\texttt{var } \mathit{front}_{S,n} : \mathit{Int}\end{aligned}$$

where n is the number of steps in the sequence specifier S. We'll use these variables to impose an order on the sequence steps later. Additionally, if the i-th step of S is a Wait instruction, we declare

$$\texttt{var } \mathit{wait\text{-}indicator}_{S,i}$$

These can later be used to find out where we should emit `Wait` actions.

With this we have all the variables we'll need.

4.4 Constraints

Now we need to encode constraints over our variables. We want to make sure that no device's input requirements and limits are violated. Furthermore, we want to ensure that all devices only move along intended paths, which may either be the prescribed ones for regulators and controllers, or, for consumers, the ones allowed by the specified transitions. Within those consumer transitions, we also need to make sure to follow the orderings imposed thereby.

First, we constrain every variable we just declared to be greater than or equal to 0. As we'll see, there is nothing else in the constraints to anchor the solution to any particular point on the number line. Constraining our sequence to start at 0 allows us to express optimization goals in absolute terms. This is vital for certain goals. For instance, if we wanted to make sure that some consumer C reaches a state S as soon as possible, we can simply minimize $\mathit{enter}_{C,S}$. If we wanted to express this in relative terms, we'd need know the first events of the sequence beforehand, or introduce a special variable indicating the beginning of the sequence. Beginning at 0 also gives us small numbers in our solution, which makes it easier to read.

We'll start with the wires. For every wire w, we assert that:

$$\mathit{edge}_w = \mathit{edge}_{\mathbf{start}(w)} \wedge \forall (d, i) \in \mathbf{ends}(w) : \mathit{edge}_w = \mathit{edge}_{d,i}$$

This means that all pins connected to the same wire change at the same time. Furthermore, if w has any monitors, we want to make sure that at least one is ready when w changes. For this we'll make use of a new meta function:

Function Name	Description
monitors(w)	Returns a list of triples indicating the device, pin and monitor type of every monitor for wire w.

With this, we specify the monitor constrains like so:

$$\exists (d, p, t) \in \mathbf{monitors}(w) : \mathit{egde}_w \geq \mathit{monitor\text{-}ready}_{d,p,t}$$

This takes care of the wires.

For the devices, we first need to generate the paths. For each device d, we generate a list of paths, each path being a list of states. Each list of states describes an allowed path through the device's states, starting $\mathbf{begin}(\textit{current-state}_d)$ and ending at $\mathbf{goal}(\textit{current-state}_d)$. For regulators and consumers, we can take the prescribed paths and filter them according to the start and end states, and take out unused states. For consumers, we run a simple path enumeration algorithm on the graph formed by the states and specified transitions. We'll denote the set of paths for a device d as $\mathbf{P}_d$.

With this set of paths, we first constrain the path selector, where:

$$\textit{path-select}_d < |\mathbf{P}_d|$$

Remember that we already constrained it to be ≥ 0 beforehand.

The outputs are constrained differently depending on the kind of device. Consumers don't have outputs in the first place. Regulators can change their outputs when they enter a new state, whereas controllers are free to change their outputs at any time once they're configured. For every controller d, we first check if it's moving from On or Configured to Off. If it does, we assert that, for every output o:

$$\textit{enter}_{d,\mathsf{Off}} > \textit{edge}_{d,o}$$

This means all output changes must happen before the controller is turned off. If it is moving from an unconfigured state to a configured state, we require that the controller is configured before we change its outputs:

$$\textit{enter}_{d,\mathsf{Configured}} < \textit{edge}_{d,o}$$

Otherwise, we don't constrain the outputs at all. Either the controller is Off or Powered throughout, in which case the outputs are always 0 and there are no transitions, or the controller is always at least Configured, meaning the outputs can change at any time.

For regulators, we have to take the paths into account. For the i-th path p_i and output o, we collect a set of states $\mathbf{S}$, where each state is in p_i and is a state where o assumes the goal value. With this, we assert that, if we're on path p_i, o changes as soon as one of those states is reached:

$$\textit{path-select}_d = i \implies \textit{edge}_{d,o} = \min_{s \in \mathbf{S}} \textit{enter}_{d,s}$$

Next, we need to make sure that all devices go through their paths correctly. For this, paths are further divided into steps. A step is simply a pair of states that are consecutive in a path. What we assert about steps is different for consumers compared to regulators and controllers.

We'll look at regulators and controllers first. For each device d, we get the paths p_i and extract their steps. Each step of a path p_i consists of a **from** and a **to** state.

First, if the step goes from Powered to Configured or vice versa, we assert:

$$\begin{aligned}\textit{path-select}_d = i \iff \textit{enter}_{d,\mathbf{from}} &< \textit{configure}_d \\ \wedge \textit{enter}_{d,\mathbf{to}} &= \textit{configure}_d\end{aligned}$$

Otherwise, we first make sure we don't skip this step:

$$\textit{path-select}_d = i \iff \textit{enter}_{d,\mathbf{from}} < \textit{enter}_{d,\mathbf{to}}$$

Then we need to figure out which inputs need to change to their target states in order for the device to transition from the **from** state to the **to** state. First, we compute the set **Changed**, which contains all inputs whose requirements change between **from** and **to**. We use this to assert that the transition happens exactly when all those inputs have changed:

$$\begin{aligned}\textit{path-select}_d = i \iff \textit{enter}_{d,\mathbf{to}} = &\max_{j \in \mathbf{Changed}} \textit{edge}_{d,j} \\ &\wedge \forall j \in \mathbf{Changed} : \textit{enter}_{d,\mathbf{to}} \geq \textit{edge}_{d,j}\end{aligned}$$

Then we compute the set **Avoid**, which contains all inputs which would transition the device to some other state if they were to change. We assert that these inputs only change at some later point:

$$\textit{path-select}_d = i \iff \forall j \in \mathbf{Avoid} : \textit{edge}_{d,j} > \textit{enter}_{d,\mathbf{to}}$$

This concludes the path constraints for regulators and controllers. For consumers, we need to take the explicitly specified transitions into account. For a consumer d, we'll again go through each path p_i, and divide it into steps. Each step being comprised of a **from** and **to** state. First we get the transition specifier S that corresponds to the step. We'll denote the length of S as n. We assert that the specified transition is completed, and that it transitions the device to the target state:

$$\begin{aligned}\textit{path-select}_d = i \iff &\textit{enter}_{d,\mathbf{from}} = \textit{front}_{S,0} \\ &\wedge \textit{enter}_{d,\mathbf{to}} = \textit{front}_{S,n} \\ &\wedge \textit{enter}_{d,\mathbf{from}} < \textit{enter}_{d,\mathbf{to}}\end{aligned}$$

Next we encode the ordering imposed by the transition specifier. We look at each step of the transition specifier separately. For the jth step in the transition specifier (starting at 1), we first check what kind of step it is. If it calls for a `Wait` action, we assert the following:

$$\begin{aligned}\textit{path-select}_d = i \iff &\textit{front}_{S,j-1} < \textit{wait-indicator}_{S,j} \\ &\wedge \textit{front}_{S,j} = \textit{wait-indicator}_{S,j}\end{aligned}$$

This makes sure the wait indicator is set correctly, and imposes the right ordering on the fronts.

Otherwise, the jth step consists of a set of inputs that need to change. We'll call this set of inputs $\mathbf{Edges}_j$. We assert that all these inputs change after the preceding front, and before the next front:

$$\begin{aligned}\textit{path-select}_d = i \iff &\forall k \in \mathbf{Edges}_j : \textit{front}_{S,j-1} < \textit{edge}_{d,k} \\ &\wedge \forall k \in \mathbf{Edges}_j : \textit{edge}_{d,k} \leq \textit{front}_{S,j} \\ &\wedge \textit{front}_{S,j} = \max_{k \in \mathbf{Edges}_j} \textit{edge}_{d,k} \\ &\wedge \textit{front}_{S,j-1} < \textit{front}_{S,j}\end{aligned}$$

Note that we also make it clear that the next front happens exactly when the last input changes, and that the fronts are ordered.

This takes care of consumers. Next we need to encode when a device's monitors become ready.

For a device d, we first check if it has any monitors. If it doesn't there's nothing to be done. If it does have monitors, we next check if it will require a `Configure` action. If no `Configure` action is needed, we consider the monitors to be always usable. If a `Configure` action is in fact required, we assert that the monitors are only available once the device has been configured. For each monitored pin p and monitor type t, we assert that:

$$\textit{monitor-ready}_{d,p,t} = \textit{configure}_d$$

.

Finally, we need to encode the global constraints given in the platform sequence specifier. Recall that these constraints establish an ordering between two sets of event, where the events are devices entering states. Take a constraint C, consisting of a **Before** set, and an **After** set. The two sets contain pairs of device names and states. The constraint expresses that the devices in the **Before** set should all reach the corresponding states before any of the devices in the **After** set reaches its assigned state. We'll declare a new variable:

$$\texttt{var}\ \ \textit{constraint-front}_C : \mathit{Int}$$

Like all other variables, it is constrained to not be negative. Additionally, we order the **Before** and **After** sets around it:

$$\begin{aligned} &\forall (d, s) \in \textbf{Before} : \textit{enter}_{d,s} < \textit{constraint-front}_C \\ \wedge &\forall (d, s) \in \textbf{After} : \textit{constraint-front}_C \leq \textit{enter}_{d,s} \end{aligned}$$

4.5 Optimization Objectives

For sequences, we don't provide any objectives out of the box, but we simply encode the objectives given in the platform sequence specifier. Each such objective describes a device d and a state s, with the objective being to have device d reach state s as soon as possible. We pose the following for each objective:

$$\texttt{minimize}\ \ \textit{enter}_{d,s}$$

4.6 Recovering a Sequence

Once the solver has solved our instance, we need to extract a sequence of action as described in section 2.5. We can extract each kind of action separately.

4.6.1 `Set` actions

For `Set` actions, we go through each controller d. We compute the set of outputs that have changed between that start and goal platform states. For each such output o, we get the value of $\textit{edge}_{d,o}$, and form the following sequence step:

$$(\textit{edge}_{d,o},\ \texttt{Set}\ \ d, o = \textbf{goal}(\textit{output}_{d,o}))$$

4.6.2 `Configure` actions

First we check which devices need `Configure` actions. We go through the devices, and check their *path-select* variable. We then check if there is a configuration step on the selected path. Then we first extract the value of the corresponding *configure* variable, to find out when it should happen. To figure out which internal variables to set, and to what value, we compare the device's internal variable assignments in the start and end platform states. This step also sets the alert limits, which can also be read from the target platform state.

4.6.3 `Wait` actions

For `Wait` actions, we check each consumer's path selector, and look on the selected path for `Wait` specifiers in the transition specifiers. If we find one, we check the corresponding *wait-indicator* variable. This together with the wait time specified in the transition specifier forms the `Wait` action.

4.6.4 `Monitor` actions

To find the `Monitor` actions, we check each wire. If the wire changed, and its goal value is not 0, we emit a `Monitor` action to monitor it. To figure out the time of the monitoring, we extract the value of *edge* of the wire, and add 0.5. This ensures that the monitoring happens before any other action that might rely on the wire's value having changed. By adding 0.5 monitor actions are grouped in between other actions, with the idea being that we can turn on a number of wires in a row, and then wait for them all to reach their target voltages, instead of alternating between enabling and monitoring.

Then we check which monitors of the wire are currently in a position to monitor it, by looking when the monitoring devices get configured. The value or range of values to wait for are found in the target platform state.

We then put all this together into a `Monitor` action.

4.6.5 Post-Processing

Once we have all our actions, we sort them according to their timestamp. We can also sort the actions within the groups given by the timestamps. Here we order `Monitor` actions with `binary` or `value` monitors before those with other types. The resulting sequence can now be handed over to some lower level system to execute it.

4.7 An Alternative Approach

There is another approach to sequence generation. First, we would take the state generation problem, and replace every variable by a function that represents the value of that variable over time. This would define a series of platform states, indexed by time. We could then relate consecutive states to each other by encoding the effects of sequence actions. These effects would be conditioned on which action is chosen for a given time step. We would then assert that the state at time 0 corresponds to the starting state of our sequence, and that the state at time *t-end* corresponds to our goal state. This approach is much more flexible. For instance, it could cope with state requirements that are arbitrary

expressions instead of ranges. It could generate sequences where a wire changes an arbitrary number of times, if necessary. Furthermore, it would be readily apparent that every step of the sequence respects the input limits and state requirements of every device, since they are explicitly asserted at every step. However, this approach is too much for the OMT solver to handle, at least as far as we have explored it.

5 Modelling Enzian

Most devices on Enzian are voltage regulators or clocks that fit neatly into the regulator concept of our model, and can be described in a rather straightforward manner. Refer to section 2.2.1 for an example. Similarly, the ispPAC controllers in the system are described as controllers. The CPU and FPGA are described as consumers. The sequence specifiers are taken from the datasheets. The FPGA is described using two states, On and Off. For the CPU, we use the states On, Reset, and Off.

There are some controllers and regulators that warrant a closer look. We also included the full description of Enzian we used in the appendix section 10.1.

5.1 The BMC

Generally, whoever is executing the sequence doesn't necessarily need to be part of the description. If the BMC controlled the platform purely through other controllers, we wouldn't need to model it at all. But since it has a few control wires on its own, we need to have a device that controls them. Because the BMC doesn't control its own supply, we model it as a controller with a single state, which is the Configured state. This means that the BMC is always on and its outputs can always be changed.

5.2 IR3581

The IR3581 regulator [16] is more complicated than the other regulators on Enzian. It's a dual loop controller, meaning it has two outputs that are configured and controlled independently, essentially acting like two regulators in one chip. It has to be configured three times: First the device itself needs to be configured to run in the correct mode, then each loop has to be configured. We model this regulator with two regulator classes, `IR3581-Parent` for the device itself, and `IR3581-Loop` for the loops. We use one instance of `IR3581-Parent`, connected by a control wire to two instances of `IR3581-Loop`. Once the parent device is configured, it asserts the control wire, allowing the loops to enter their Powered state. This signifies that the loops are now ready and can themselves be configured.

5.3 ISL6334

This regulator [17] is different in that its output voltage level is not controlled by sending a command via a bus. Instead, the device reads the value of a set of dedicated pins and sets its output voltage based on that. We abstract away this distinction and model it just like the other regulators, modelling the pins as an

internal value instead. We leave it to a lower layer to know that the ISL6334 requires a GPIO write instead of a bus message.

5.4 Clocks

The clocks [18] are modelled just like voltage regulators, except that their outputs are `clock` pins instead of `DC` pins.

5.5 INA226

Enzian also contains a few INA226 chips [19]. These exist purely to monitor other wires, so they're modelled as controllers without outputs. They get inputs with monitor and alert specifiers for the wires they monitor. Since they're neither supplying something nor controlling other devices, we need to explicitly request them to be turned on in a platform state specifier if we want to use their monitoring capabilities. If we don't require them to be running, the solver might decide they're not needed.

6 Implementation

The state and sequence generation procedures were implemented in Guile scheme, using Z3 as the OMT solver. We feel that scheme is a good language for this task because it is a high level language that works well for symbolic computations. Additionally, both Scheme and SMTLib, the input language for Z3, are based on S-expressions, which makes it very natural to interact with the solver in SMTLib.

Due to limitations in SMTLib and Z3, some of the constructs used in sections 3 and 4 need to be encoded a little bit differently. Namely, Z3 doesn't support optimization together with quantifiers, therefore every quantified assertion we laid out in sections 3 and 4 is fully instantiated beforehand, making our OMT instances quantifier-free. Furthermore, since SMTLib doesn't provide minimum and maximum functions, we define our own *min* and *max* functions of two arguments. During instance generation, min and max over collections are then expanded using these.

Furthermore, during the limit generation step of the state generation, we don't solve for every objective at once. Instead, we solve for each objective one after the other. However, since the objectives are independent, it is possible to solve each individually. To make this fast, we assert our constraints only once, and solve each objective in a new assertion scope. This allows Z3 to retain facts about our constraints, and reduces the traffic between our program and Z3.

The implementation works on description files. These are text files containing class descriptions and platform descriptions, in a structure that is similar to the one laid out in section 2.

These files are consumed by two programs, `find-sequence` and `find-state`. Both take as arguments a description file, a platform name, and one or more names of either a sequence or state specifier. `find-state` generates each state given, and saves it in a state description file. These are used by `find-sequence`, which in turn saves the sequences it generates in sequence description files.

There are two reasons why `find-sequence` uses files instead of just calling

`find-state` to find the start and end states of each sequence. First, it's faster to just generate the states once and then refer to stored version when generating sequences. Second, say we wanted to have one sequence going from some platform state A to some state B, and another from B to C. We would want to be sure that both sequences use the exact same B. Otherwise we'd have a sequence from A to B, and another from B' to C, with no way of going from B to B'. However, if B is read from a file both times instead of being generated anew, we can be sure that the same B is used for both sequences.

The programs are written such that other OMT solvers could be added with little effort. We support OptiMathSat [4] for sequence generation, but it doesn't have some of the features we use in state generation. It would be possible to target an older version of SMTLib, by using integers in place of symbols for device states. However, having symbolic names is more convenient for debugging purposes, so it was decided to keep them.

Both programs have the same basic workflow. The program reads the necessary files, and starts an interactive session with the solver. The OMT instance is generated and fed to the solver. If it is found to be satisfiable, we ask the solver for the assignments to the variables we need to describe the resulting sequence or state. We also write out a full transcript of the interactions with the solver. This is mainly useful for debugging, and also to measure the time it takes to solve the OMT instance. This lets us evaluate the time it takes to solve separately from the time it takes to generate the instance.

To further aid debugging, we can run the solver with unsat-core generation turned on. This lets us see which assertions conflict when we generate an unsatisfiable instance. Using unsat-core effectively also requires equipping each assertion with a name. The programs do that when we run them with unsat-cores enabled. Unfortunately, requiring the solver to produce unsat-cores can occasionally make it take much longer to solve. For this reason, unsat cores are disabled by default.

Since state generation can take a couple of seconds, we added an option to search for multiple states in parallel, where each state search runs with a separate instance of the solver.

6.1 Simplifications

Some features described in section 2 were left out of the implementation, because they aren't needed to model Enzian. The implementation doesn't support monitoring of clock signals, save for `binary` monitors. So the `frequency` monitor type is not supported. Raising alerts on `clock` pins is also not supported.

We also allow the output accuracy to be omitted from an output specifier, in which case a default accuracy applies that is defined at the platform level.

6.2 Sequence Translation

In order to be able to run a generated sequence on actual hardware, we need to turn a sequence description file into something that can run on the BMC. We translate a sequence into python code, which makes use of functionality provided by the power management software already present. To do this translation, the `translator` program was created. This program takes as input a sequence description, and a translation specification. This specification is a col-

lection of function definitions and pairs of patterns and code templates. The patterns are matched against the steps of the sequence. If a pattern matches, the corresponding code template is instantiated. The templates may refer to functions defined in the specification, and to variables in the pattern. This split into a generic match/instantiate engine and a specific translation specification should make it easier to write translations into other formats as well.

Unfortunately, due (presumably) to liberal use of captured environments together with dynamic compilation and evaluation, it is rather slow. It takes about half a minute to translate the full Enzian power-on sequence, which only consists of about 80 steps. We think this is acceptable, since it's not really a part of the state and sequence generation, and could easily be improved by writing a dedicated translator tailored to a specific output format.

7 Evaluation

We evaluate our system on three fronts. First, we investigate if our implementation can handle larger platforms in a reasonable time frame. Then, we want to see if our optimization goals make sense. For this we generate states and sequences for the Enzian platform, and compare them to the states and sequences used in the power management software that's in use on Enzian currently. We'll refer to these as the default states and sequences.

We'll compare the time it takes to turn on the platform, as well as the power transmitted over selected wires.

7.1 Scaling

For this experiment we want to see how our implementation scales as the size of the platform increases. A program was created that generates a platform description with a given number N of CPUs. It essentially copies everything in the Enzian power tree that supplies the CPU, including the ispPAC sequencer assigned to the CPU side. The only devices that are not replicated are the BMC, the PSUs, and the main clock. Using this program, we can generate platforms of different sizes. Each platform we generate has a platform state specifier specifying the all-on state, in which all N CPUs are on. Additionally, each specifies the sequences power-on-all and power-off-all that transitions from initial to all-on, or from all-on to initial, respectively. We run the state generation programs for all-on and initial, as well as the sequence generation program for power-on-all and power-off-all. Additionally, we run Z3 on the SMTLib transcript of each of these runs, to get an idea of the cost of generating the OMT instance as compared to solving it.

These experiments were run on an Intel® Core™ i5-4210M running at 2.60 GHz, equipped with 8GB of RAM. The software used was GNU Guile 3.0.7 and Z3 4.8.10. The whole procedure was repeated three times, and all measurements averaged. The timings are shown in figure 6. We can see that state generation is generally more expensive, and its scaling is worse compared to sequence generation. We go from 10s for 4 CPUs, to two minutes for 16, up to 10 minutes for the full set of 32 CPUs. However, considering the full 32 CPU platform contains a total of 488 devices with 588 wires, we argue that this is still reasonable.

We also see that it takes more time to find the all-on state compared to the

initial state. This makes sense, since the initial state is more constrained, making it easier for the solver to find an assignment. For the sequence generation, we see that it's generally fast. We also see that finding how to turn the platform off takes longer than finding how to turn it on. This is likely due to the fact that regulators can be turned off in multiple ways. For instance, we might turn off an enable signal first, or we might just cut the regulators power supply. When turning it on, we only allow one way, which makes it easier to find the whole sequence.

We also see that the time it takes to solve power-off sequences is not monotonically increasing, primarily due to the time it takes to solve the OMT instances. We have not investigated the cause of this, but suspect

In general, we can see that the overhead for generating the OMT instances are reasonable. Unlike the OMT solving, the cost of generating the problem doesn't appear to be influenced by which state or sequence we're looking for.

7.2 Startup Speed

We compared how long it takes to turn on an Enzian using the sequence we generated. We compared it against how long it takes the current power-up script to do the same. These tests were ran on zuestoll-08. We added two functions to the power management software that either run the default sequence, or our generated sequence. Then they report the time it took for the sequence to complete and shut down the platform again.

For each measurement, we first restart the power service and start a new power management shell, to ensure the same conditions for each run. We discovered that the power service currently has an issue where consecutive power cycles take longer each time, which is why we restart it in between.

Then we run one of the functions, which will start up the platform and report the time. The platform is the shut down again.

We found that our power-on sequence is slightly slower than the default one, taking around 8.3s instead of 7.5 for the default one.

There are a couple of factors to be aware of here. First, it seems reasonable that the difference is small. After all, both sequences do the same things, they just do it in a different order. Second, we might have expected the generated sequence to run faster. After all, it tries to fit in other operations in between turning on regulators, and waiting for them to reach their target value. However, a closer examination reveals that most regulators come up very quickly. In fact, all voltage regulators reach their target values so quickly that both sequences only need to make one or occasionally two measurements until the output has reached the target voltage. However, the three clock generators are another story. The main clock generator takes upwards of one second to stabilize, and the clocks that depend on it can easily add another half a second on top of that. The default sequence cleverly configures the clocks immediately after the sequencers, followed by most of the other devices. Only then does it start waiting for the clocks to stabilize. Our model falls short in that it doesn't know that it should maximize the time between configuring and checking the clocks. This is due in part to the fact that we can't specify that, within a set of actions that are not ordered relative to each other, configuring the clocks should come first. It is also due to the rather simplistic mechanism used to decide when to start waiting for a value. In theory, we could squeeze in a few more actions before we actually

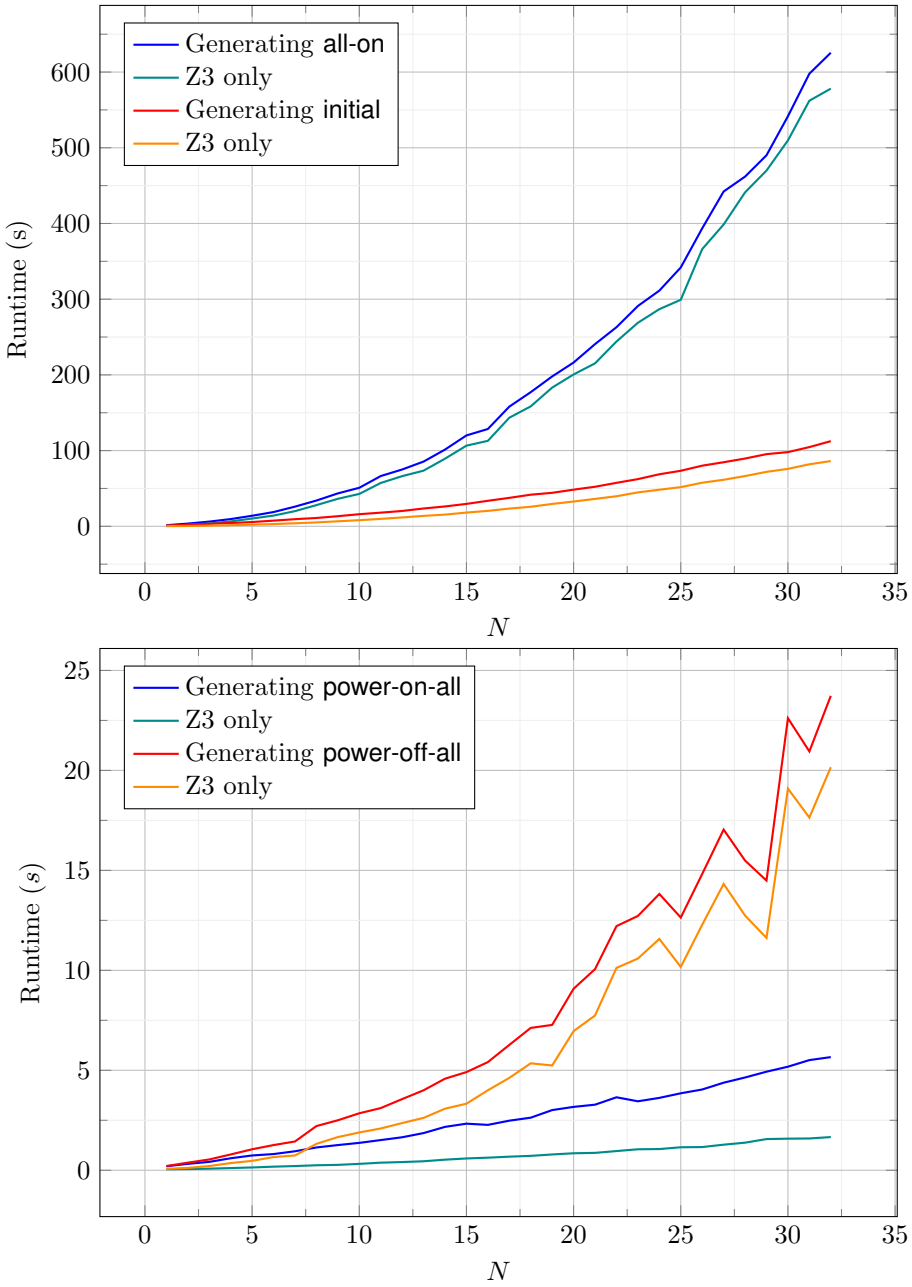


Figure 6: The top plot shows the runtime of state generation for different numbers N of CPUs. It shows the time to generate the states all-on and initial, as well as the time it took to solve just the OMT instances. The bottom plot shows the time it took to generate the sequences from initial to all-on and back.

have to have the clocks stable. We might argue that recognizing that the clocks should be configured first is an optimization that can be left to the lower layer, since we generally want to allow it to take advantage of more detailed knowledge and reorder things the order of which we declare to be irrelevant.

7.3 Power Consumption

In this experiment we want to assess if the optimizations we apply to power states actually can be effective in reducing the power consumption. We compare a generated platform state against the state that's implemented in the current power management software on the Enzian BMC. Again, we'll call this the default state. We focused on the wires VDD_CORE and VCCINT_FPGA. VDD_CORE is the main supply for the CPU, and VCCINT_FPGA. The default state sets these wires to 0.95V and 0.9V respectively. In our generated state, VDD_CORE is set to 0.947, and VCCINT_FPGA is set to 0.88V. Note that due to the limited register size in the regulators, these values might get rounded up or down a bit. We did confirm that the VDD_CORE voltage was indeed set to a different value compared to the default state.

We want both devices to be under load and consume some power. On the CPU, we ran the on board powerburn mode at 65%. For the FPGA, we used a powerburn bitstream. This bitstream consists of multiple power burning blocks which can be switched on independently. We ran it with 18 out of 24 blocks enabled. The reason for this is that the FPGA starts throttling if too many blocks are turned on, which might interfere with our measurements.

We measured the current and voltage on these wires using the current monitors of the associated regulators, and the voltage monitors of the associated ispPACs. The measurements were collected using a logging script developed by Martsenko [20]. For each measurement, we first power the platform on, using either our generated sequence, or the regular one that is already on the BMC. Then, the power burn on the device under test is started. The logging script to collect the measurements is started and left running for approximately one minute, at which point it is stopped and the whole machine turned off. We then let the regulators cool down for a bit before starting the next run.

The measured voltages and currents are then averaged, and multiplied to compute the power. The resulting values can be seen in figure 7. For the FPGA, we get a reduction of about 4W, or about 5%. On the CPU side, we see a slight reduction. Given how little opportunity we have to lower this voltage, this is to be expected.

We also measured the DRAM supply wires. For the CPU, we ran a builtin memory test called 'Random Data', since we have found that test to have a comparatively high and consistent current draw. The CPU side has 2 times 32GB of RAM. We configured the test to use all 48 cores. While the test was running, we collected measurements using the logging script.

The FPGA was programmed with a memory test bitstream [21]. We found that just programming this bitstream already increased DRAM voltages sufficiently for our purposes, so no further configuration was done. Once the FPGA was programmed, we again collected measurements for about one minute.

It should also be noted that the DRAM measurements on the FPGA side were performed on a different machine than the one where the other evaluations ran, due to the latter having issues with the FPGA DRAM. The FPGA has 4

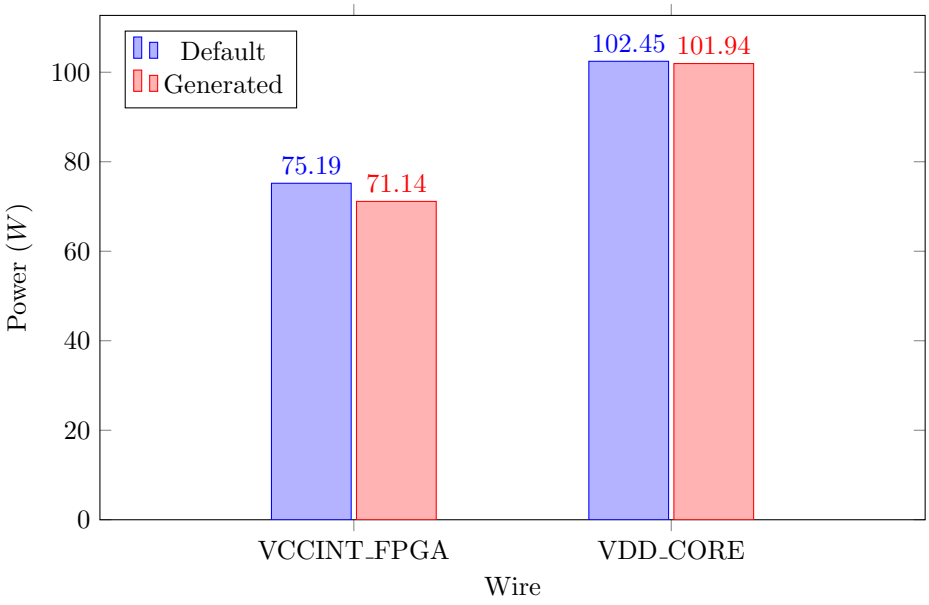


Figure 7: Shows the power delivered by VCCINT_FPGA and VDD_CORE. Both were measured while the associated device was under a power burn workload. The *Default* bars show the power used with the default power state, and the *Generated* bars show the power used with our generated state.

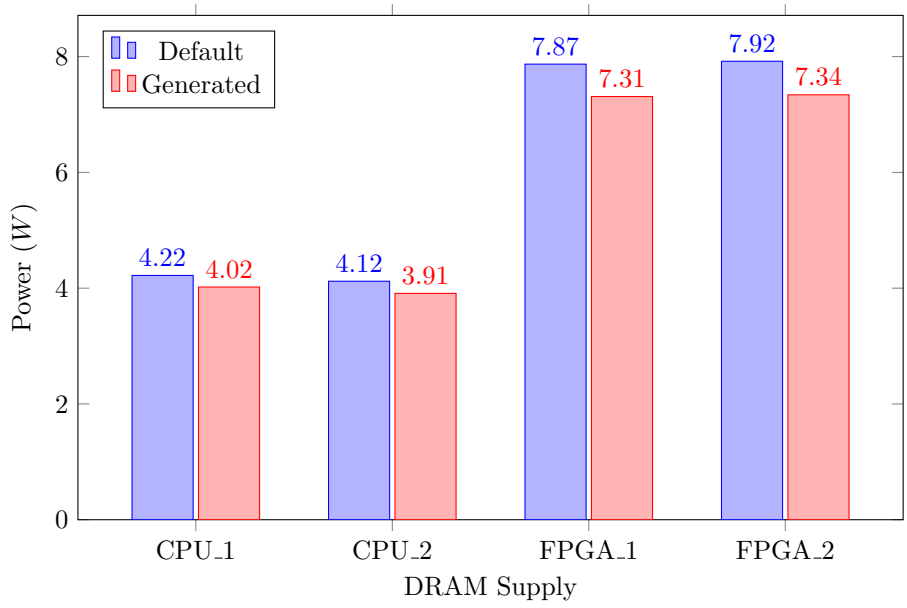


Figure 8: Shows the power delivered by the main DRAM supply wires while running memory tests under the default state, shown by the *Default* bars, and our generated state, shown by the *Generated* bars.

128GB LRDIMMs.

Both of these procedures where carried out in the default platform state, and the one generated by our tool. The resulting voltage and current measurements were averaged and used to compute the power figures. Since both the CPU and FPGA have two sets of DRAM supplies, both main supply wires on each side have been measured. The results can be seen in figure 8. The two DRAM supply wires on the CPU side have been labelled as CPU_1 and CPU_2, with the wires on the FPGA side being labelled analogously.

We can see a reduction of around 200mA, or roughly 4.7%, on the CPU side, and a reduction of 560 to 580mA on the FPGA side, or around 7.7%.

We collected additional statistics about the measurements in figure 10. It shows that both voltage and currents are mostly stable, and deviating little from the average value. The exception being the currents in the CPU DRAM measurements. We have plotted the voltages and currents for one of the CPU DRAM wires over time in figure 9. As we can see, the current in both the default and generated state is still relatively stable, with occasional drops. It's not clear where these drops come from, but they seem to affect both states, so we don't believe they're caused by something in our generated state.

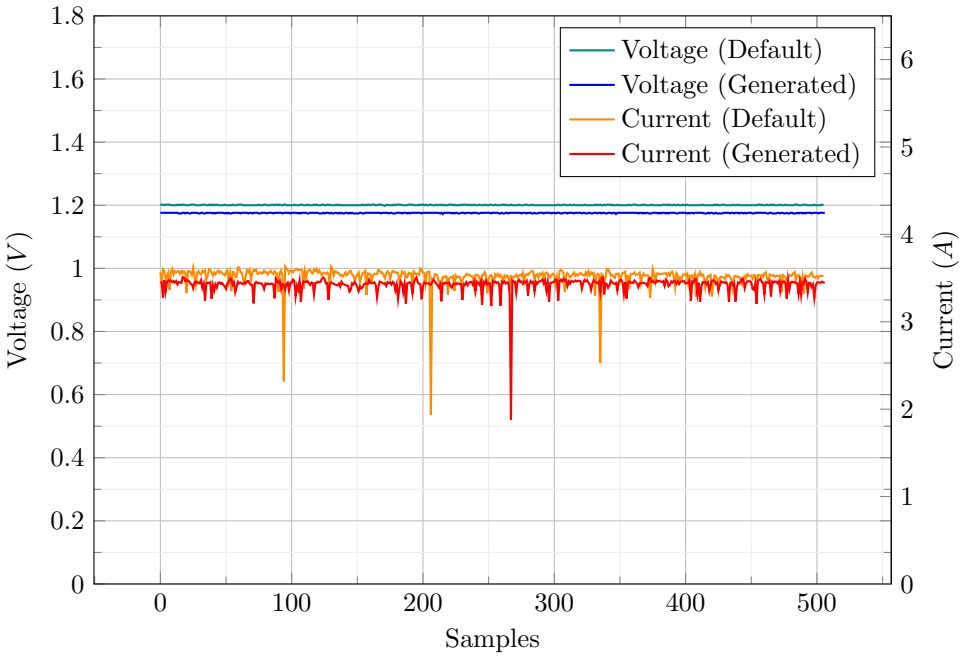


Figure 9: The voltage and current over wire CPU_1 over time.

7.4 Conclusion

We have shown that our approach can work with large platforms at speeds that are acceptable for off-line use. However, we fall short when it comes to generating faster sequences, with the key weakness being how we decide when to start waiting for devices. As far as power states are concerned, we have shown

that we can reduce power on selected wires. However, the potential for power savings are constrained by how flexible a device is with the range of voltages it accepts.

Wire	State	Unit	Min	Avg	Max
VDD_CORE	Default	V	0.968	0.970	0.972
		A	104.750	105.601	106.000
		W	101.398	102.452	103.032
	Generated	V	0.968	0.970	0.972
		A	103.750	105.091	105.750
		W	100.430	101.941	102.789
VCCINT_FPGA	Default	V	0.932	0.934	0.936
		A	80.500	80.500	80.500
		W	75.026	75.189	75.348
	Generated	V	0.910	0.913	0.914
		A	77.500	77.930	78.000
		W	70.525	71.143	71.292
CPU_1	Default	V	1.198	1.201	1.202
		A	1.822	3.517	3.636
		W	2.186	4.222	4.371
	Generated	V	1.172	1.175	1.178
		A	1.880	3.419	3.517
		W	2.211	4.019	4.136
CPU_2	Default	V	1.200	1.203	1.206
		A	1.624	3.421	3.530
		W	1.952	4.116	4.250
	Generated	V	1.174	1.178	1.180
		A	1.728	3.322	3.433
		W	2.032	3.913	4.044
FPGA_1	Default	V	1.205	1.207	1.209
		A	6.482	6.525	6.538
		W	7.820	7.876	7.898
	Generated	V	1.160	1.163	1.165
		A	6.235	6.290	6.317
		W	7.241	7.313	7.353
FPGA_2	Default	V	1.205	1.207	1.210
		A	6.517	6.562	6.577
		W	7.863	7.921	7.948
	Generated	V	1.160	1.163	1.165
		A	6.265	6.315	6.332
		W	7.278	7.342	7.369

Figure 10: Shows the minimum, average, and maximum values attained by different wires during our measurements.

8 Limitations and Future Work

Better Encoding of Monitoring As we've seen in the evaluation, our current scheme of placing a monitoring action just after the associated wire changes can lead to those monitoring actions happening sooner than necessary. This could be remedied by introducing variables corresponding to the time step where a wire is monitored. Such a variable would be constrained to occur after the wire itself changes, and before anything that depends on that wire changing.

Stability Testing Ideally, our states and sequences should be subjected to a number of tests to confirm that the platform stays stable under a variety of loads, and everything still operates correctly. We have merely shown that we can turn on an Enzian, and that it doesn't crash under a certain types of loads.

Verification It would be reassuring to know with more certainty that our platform states and sequences are going to respect all the semantics of our platform descriptions, like sequencing requirements and voltage limits. The correctness of the OMT solver will likely need to be presupposed, but it would be good if we could show that the generated SMTLib code correctly encodes every constraint imposed by the description.

Better Handling of Abstraction Gaps On occasion, we discover that our model does not match reality. For instance, during our tests on Enzian, it was discovered that the wire UTIL_3V3 is more sensitive to undervolting than our description implies. It's not clear to us why exactly this is, and the exact voltage where problems start happening seems to also depend on the particular machine. If such a problematic wire is found, the solution at present is to set the wire to a fixed voltage in the platform state specifier, and regenerate all states and sequences. It would be nice if this was more automatic. At the low end, this could just mean that we can tell the program that the voltage was too low, and it would figure out a better value on its own. With a bit more work, me might just tell the program which kind of fault was generated by the system, and figure out what to change based on that. At the other end of the spectrum, the generated sequence would intercept the error, adjust the target state, roll back sufficiently far, and continue the sequence with a corrected value. This ties in to the next item:

Speed Improvements At present, the state generation, and to a somewhat lesser extent the sequence generation, are too slow to be used in an online scenario, where we would want to react dynamically to a changing platform. It would be nice if we could generate states and sequences faster. For this, we would need to improve both the generation of the OMT instances, as well as the solving time of the instances themselves. We might think about generating states that are derived from similar states, with most of the variables already assigned. The OMT instances for such problems would hopefully be easier to solve. A technique like this might apply in scenarios where we need to react to a fault by adjusting a voltage.

There may be other gains to be had by optimizing our OMT encoding, or by tuning the solver settings. There are also a host of improvements to

be made with the programs themselves. The algorithms and datastructures used could certainly be improved. We should also consider choosing a different implementation language, preferably with Z3 bindings available. Z3 bindings would avoid the overheads associated with generating, transmitting, and parsing SMTLib.

Tuning Optimization Objectives . So far we've worked with a fairly simple optimization objective: Minimizing all voltages. It would make sense to investigate different schemes. For example, we might only minimize voltages on wires with a lot of current, and try to steer low power wires towards the middle of their acceptable range, hopefully improving stability at a minimal increase in power. In sequences, we could try to group `Set` commands to an ispPAC in a way that allows a lower layer to fuse these commands into a single SMBus transaction. We might also try and come up with objectives to lower the power consumed over the course of a sequence. For instance, instead of turning on consumers as soon as we can, we could try and start turning on each consumer as late as possible, so that all consumers come online at the same time. Many alternative objectives require model extensions:

Model Extensions There are a number of aspects that we might want our model to be informed about. A significant one is modelling currents. Modelling currents could allow for more sophisticated power modelling, and might allow us to come up with limits for devices that can raise alerts on currents. Current modelling has a number of issues to be addressed. For one, modelling currents accurately likely requires empirical data gathered from a running platform. A datasheet will usually just provide a maximum and a typical draw, which may not be accurate enough to get the most out of the optimization. Furthermore, including variable currents might make our OMT problem non-linear, which would make it harder to solve.

We may also want to model the efficiency of regulators. This also opens up different optimization avenues. For instance, we could search for states where every regulator operates as efficiently as possible, instead of focusing on lowering power consumption.

Another extension might be to make the model more aware of the communication capabilities of devices. Take the ispPACs for instance, which may set multiple outputs in a single SMBus transaction. If the model were aware of this, we might use it to optimize sequences. It should be noted that this can already be done to an extent. The lower layer interpreting the sequence action knows which `Set` actions to the ispPACs are not ordered relative to each other, and could fuse them. The sequence generator just makes no effort to come up with sequences where many `Set` commands may be optimized like this.

Change Sequence Generation We might want to be able to compute sequences that don't fulfill the assumptions we laid out in section 4. One way to do this might be to revisit the alternative approach described in section 4.7. Originally, we only constrained the start and target states using the constraints laid out the corresponding state specifiers. This means the solver not only has to find the sequence, but also has to find the optimal start and target states. Fixing the states first and then using the other approach would be the first

optimization to be tried, although it's likely that more would be required. The advantage would be that it might be possible to simplify how we model devices, and that we would be able to look for sequences we can't generate with the current mechanisms.

Hierarchical Platforms It may be possible to abstract a whole platform as a single consumer. The platform's states would become the consumer's states, and the platform's sequences would be turned into the consumer's sequences. Applications for this might include modelling platforms with identical, independent substructures, or ensembles of largely independent platforms controlled by a central controller.

While this is possible now, abstracting sub platforms as consumers would likely be more efficient, since it would allow us to avoid solving the same subproblems multiple times, and we'd be solving multiple smaller OMT instances instead of a single large one.

For example, a rack of Enzians with a rack controller might be modelled this way. We'd probably also want to change something about how platform states work, to avoid having to explicitly generate every combination of machines being on or off.

Profile Guided Optimization In order to increase the quality of our states and sequences, it might be helpful if we could generate optimization objectives from measurements of the platform. For instance, if we had a set of measurements about how long each regulator usually takes to reach its target, we could use that to make sure that the regulators that take the longest get turned on as soon as possible, and that we only start waiting for them as late as possible.

Sequence Graphs At the moment, a sequence is given to the lower layer as a linear sequence of sets of steps. This does give us some freedom to reorder steps within sets, but it still imposes ordering constraints that are not real. The Enzian platform is a good example for this, since the power tree on the CPU side is more or less independent of the power tree on the FPGA side. At the moment, we can have situations where our progress on the FPGA side is blocked because we're waiting for some regulator on the CPU side, or vice versa. If sequences were output as directed acyclic graphs, which is what they really are, the lower layer that implements it would have more freedom to react to dynamic conditions.

For instance, if the clock on the CPU side takes a little longer to lock onto its frequency, the lower layer could continue turning on the FPGA if it knew the real dependencies. As it stands, the lower layer is forced to wait because it does not know which of the subsequent steps actually depend on the CPU clock and which don't.

Cooperating Executors Our model assumes that a single unit will be responsible for executing the sequence. On Enzian, this is the Enzian BMC. However, Enzian is also equipped with two power sequencers, the ispPACs. They are programmable, and in theory, parts of the power sequence could be performed by them. A possible avenue to address this might be implementing

the hierarchical platform idea mentioned above. We could then model everything under the control of an ispPAC as a separate platform, which can then be abstracted as a consumer to be controlled by the BMC. However, there may be other solutions that are orthogonal to hierarchical platforms. It may also be possible that this could be done just by taking the generated sequence, and splitting it up among multiple controllers, with the necessary synchronization added.

9 Conclusion

We have seen how we can abstract a modern computing platform's power infrastructure by describing its devices and their connections in a declarative model. From this model, OMT problems can be generated that encode the model's semantics, and can be used to find power states, and sequences to transition between them. The optimization capabilities of OMT solvers can be used to look for power states and sequences that are optimal with respect to some objective, like having low voltages throughout the system.

We demonstrate that this approach is reasonably performant. Also, while the sequence we tested is not faster than the one currently in use on Enzian, the state it takes the platform to does save power, with the most significant saving we observed being the FPGA main supply, which was reduced by 4W.

References

[1] Jasmin Schult. "A model-based approach to platform-level power and clock management". ETH Zurich, 2020.

[2] Robert Nieuwenhuis and Albert Oliveras. "On SAT Modulo Theories and Optimization Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 156–169. ISBN: 978-3-540-37207-3.

[3] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 194–199. ISBN: 978-3-662-46681-0.

[4] Roberto Sebastiani and Patrick Trentin. "OptiMathSAT: A Tool for Optimization Modulo Theories." In: *Proc. International Conference on Computer-Aided Verification, CAV 2015*. Vol. 9206. LNCS. Springer, 2015.

[5] *Z3Prover*. `https://github.com/Z3Prover/z3`. Online. Accessed 2021-09-19.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2017.

[7] The Enzian Team. *Enzian: a research computer built by the Systems Group at ETH Zurich*. Online. Accessed 2021-09-19. URL: `http://enzian.systems`.

[8] Enzian BMC Power Management Tools Repository. https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bmc-powermgmt/-/tree/master. Online. Accessed 2021-09-19.

[9] Ben Fiedler. "Trustworthy BMC Software for Modern Hardware". MA thesis. 2021.

[10] Cedric Heimhofer. "Towards high-assurance Board Management Controller software". en. MA thesis. Zurich: ETH Zurich, 2021. DOI: 10.3929/ethz-b-000490635.

[11] Mirela Simonović, Vojin Živojnović, and Lazar Saranovac. "Formal model for system-level power management design". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 1599–1602. DOI: 10.23919/DATE.2017.7927245.

[12] Jasmin Schult et al. "Declarative Power Sequencing". In: EMSOFT. 2021.

[13] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 135–152. ISBN: 9781450324724. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[14] Maxim Integrated. *MAX15301. Multiphase Master with PMBus Interface and Internal Buck Converter, Document Revision 2*. Online. Accessed 2021-09-19. 2015. URL: https://datasheets.maximintegrated.com/en/ds/MAX20751.pdf.

[15] Lattice Semiconductor. *ispPAC-POWR1220AT8, Data Sheet DS1015*. Online. Accessed 2021-09-19. 2012. URL: https://www.latticesemi.com/-/media/LatticeSemi/Documents/Solutions/Packaging-Solutions/ispPAC_POWR1220AT8-Data-Sheet1015.ashx?la=en.

[16] International Rectifier. *IR3581, Dual Output Digital Multi-Phase Controller, Document Revision 2.2*. Onlune. Accessed 2021-09-19. Apr. 2014. URL: https://www.infineon.com/dgdl/pb-ir3581.pdf?fileId=5546d462533600a401535680609b28f9.

[17] Renesas. *ISL6334D VR11.1, 4-Phase PWM Controller with Phase Dropping, Droop Disabled and Load Current Monitoring Features, Document Revision 4.00*. Online. Accessed 2021-09-19. Apr. 2016. URL: https://www.renesas.com/us/en/www/doc/datasheet/isl6334d.pdf.

[18] Skyworks Solutions. *Si5395/94/92 Data Sheet, Document Revision 1.2*. Online. Accessed 2021-09-19. 2021. URL: https://www.skyworksinc.com/-/media/SkyWorks/SL/documents/public/data-sheets/si5395-94-92-a-datasheet.pdf.

[19] Texas Instruments. *INA226, High-Side or Low-Side Measurement, Bi-Directional Current and Power Monitor with I 2C Compatible Interface, Document Revision A*. Online, Accessed 2021-09-19. 2015. URL: https://www.ti.com/lit/ds/symlink/ina226.pdf.

[20] Kristina Martsenko. "Developing and Evaluating Power Models of Heterogeneous Computer Systems". en. MA thesis. Zurich: ETH Zurich, 2021. DOI: 10.3929/ethz-b-000488930.

[21] *Enzian FPGA Simple Memory Test*. Online. Accessed 2021-09-22. URL: https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-fpga-simple-memory-test.

10 Appendix

10.1 Enzian Description

Here is the full description of the Enzian platform that we used, in the form our tools work with. It is mostly analogous to the tree based presentation used in the main text.

```
(regulator (MAX15301 (preset DC))
  (input PWR DC (0 14))
  (input EN logical 1)
  (internal VID DC
    (initially preset))
  (output V_OUT DC)
  (output PGOOD logical 1)
  (monitor voltage PWR)
  (monitor voltage V_OUT)
  (off
    (requires (PWR (0 4.4)))
    (V_OUT 0)
    (PGOOD 0))
  (powered
    (requires (PWR (5.5 14))
              (EN 0))
    (V_OUT 0)
    (PGOOD 0))
  (configured
    (V_OUT 0)
    (PGOOD 0))
  (on
    (requires (EN 1)
              (PWR (5.5 14))
              (VID (0.6 5.25)))
    (V_OUT VID)
    (PGOOD 1)))
```

```
(regulator NCP
  (input VCC DC (0 6))
  (input VRI DC (0 6))
  (output VREF DC)
  (off
    (requires (VCC (0 2.3)))
    (VREF 0))
  (on
    (requires
      (VCC (2.357 5.5))
      (VRI (0.868 3.6)))
    (VREF (/ VRI 2))))
```

```
(regulator (MAX8869 (preset DC))
  (input V_IN DC (0 6))
  (input SHDN logical 1)
  (output V_OUT DC)
  ;(let limit (max (+ 0.5 preset) 2.7))
  (let limit 2.7)
  (off
    (requires (V_IN (0 limit)))
    (V_OUT 0))
  (powered
    (requires (V_IN (limit 5.5)))
    (requires (SHDN 0))
    (V_OUT 0))
  (on
    (requires (V_IN (limit 5.5)))
    (requires (SHDN 1))
    (V_OUT preset)))
```

```
(regulator (MAX15053 (preset DC))
  (inputs
    (V_IN DC (0 6))
    (V_EN logical 1))
  (outputs
    (V_OUT DC))
  (let limit 2.7)
  (off
    (requires (V_IN (0 limit)))
    (V_OUT 0))
  (powered
    (requires
      (V_IN (limit 5.5))
      (V_EN 0))
    (V_OUT 0))
  (on
    (requires (V_IN (limit 5.5)))
    (requires (V_EN 1))
    (V_OUT preset)))
```

```
(regulator ISL
  (input VCC DC (0 6))
  (input EN_PWR logical 1)
  (input EN_VTT DC (0 14))
  (output VOUT DC)
  (internal VID DC (initially 0))
  (off
    (requires (VCC (0 3.88)))
    (VOUT 0))
  (powered
    (requires (VCC (4.75 5.25)) (EN_PWR 0))
    (VOUT 0))
  (configured
    (VOUT 0))
  (on
    (requires
      (VCC (4.75 5.25))
      (EN_PWR 1)
      (VID (0.5 5.5))
      (EN_VTT (0.87 14)))
    (VOUT VID)))
```

```
(regulator (IR3581-parent (thresh DC))
  (input VCC DC (0 4))
  (input VIN DC (0 13.2))
  (output loop-enable (logical 1))
  (off (requires (VCC 0))
       (loop-enable 0))
  (powered
    (requires (VCC (2.9 3.63)))
    (loop-enable 0))
  (configured
    (loop-enable 1)))
```

```
(regulator (IR3581-loop (thresh DC))
  (input VCC DC (0 4))
  (input VIN DC (0 13.2))
  (input loop-enable (logical 1))
  (input EN (logical 1))
  (internal VID DC (initially 0))
  ; from datasheet - system accuracy up to 85C
  (output VOUT DC
    #:accuracy
    (+- (if (and (>= VID (DC 2.005)) (<= VID (DC 3.04)))
          (% 1.1)
          (if (and (>= VID (DC 1)) (<= VID (DC 2)))
            (% 0.5)
            (if (and (>= VID (DC 0.8)) (<= VID (DC 0.995)))
              (DC 0.005)
              (if (and (>= VID (DC 0.25)) (<= VID (DC 0.795)))
                (DC 0.008)
                ; dont-care value
                (DC 20000)))))))

  (monitor voltage VIN)
  (monitor voltage VOUT)
  (off (requires (loop-enable 0))
    (VOUT 0))
  (powered
    (requires (VCC (2.9 3.63)) (loop-enable 1) (EN 0))
    (VOUT 0))
  (configured
    (VOUT 0))
  (on
    (requires (VCC (2.9 3.63))
              (loop-enable 1)
              (EN 1)
              (VIN (thresh 13.2)))
    ; TODO find right values here
    (requires (VID (0.5 1.5)))
    (VOUT VID))
  (alerts voltage VOUT))
```

```
(regulator (MAX20751 (default DC))
  (inputs
    (VDD33 DC (0 4))
    (VDDH DC (0 23))
    (VR_ON logical 1))
  (output V_OUT DC
    #:accuracy
    ((if (< VID (DC 1)) (DC -0.005) (% -0.5))
     (if (< VID (DC 1)) (DC +0.005) (% +0.5))))
  (monitor voltage VDDH)
  (monitor voltage V_OUT)
  (internal VID DC
    (initially default))
  (off (requires (VDD33 0))
    (V_OUT 0))
  (powered
    (requires (VDD33 (2.97 3.63)) (VR_ON 0))
    (V_OUT 0))
  (configured
    (V_OUT 0))
  (on
    (requires
      (VDD33 (2.97 3.63))
      (VID (0.5 1.52)) ; datasheet
      (VR_ON 1)
      (VDDH (8.5 14)))
    (V_OUT VID)))



(regulator Oscillator
  (input VDD DC (0 3.6))
  (output CLK clock)
  (off (requires (VDD 0)) (CLK 0))
  (on (requires (VDD (2.6 3.6))) (CLK 50)))
```

```
(regulator SI5395
  (input VDD DC (0 3.6))
  (input CLK_IN clock  (0 50))
  (output CLK clock)
  (monitor binary CLK)
  (off (requires (VDD 0))
    (CLK 0))
  (powered (requires (VDD (2.6 3.6)))
    (CLK 0))
  (configured (CLK CLK_IN)))



(regulator PSU
  (input EN logical 1)
  (output OUT DC)
  (off (requires (EN 0)) (OUT 0))
  (on (requires (EN 1)) (OUT 12)))



(regulator Main-PSU
  (input EN logical 1)
  (outputs
    (PGOOD logical 1)
    (V33_PSU DC)
    (V12_PSU DC)
    (V5SB_PSU DC)
    (BMC_VCC_3V3 DC)
    (V5_PSU DC))
  (off (requires (EN 0))
    (V33_PSU 0)
    (V12_PSU 0)
    (V5SB_PSU 5)
    (PGOOD 0)
    (BMC_VCC_3V3 3.3)
    (V5_PSU 0))
  (on (requires (EN 1))
    (V33_PSU 3.3)
    (V12_PSU 12)
    (V5SB_PSU 5)
    (PGOOD 1)
    (BMC_VCC_3V3 3.3)
    (V5_PSU 5)))
```

```
(controller ISPPAC
  (input VCC DC (0 4.5))
  (input VCC_IN DC (0 6))
  (inputs
    (IN1 logical 1)
    (VMON1 DC (0 13.9))
    (VMON2 DC (0 5.734))
    (VMON3 DC (0 5.734))
    (VMON4 DC (0 5.734))
    (VMON5 DC (0 5.734))
    (VMON6 DC (0 5.734))
    (VMON7 DC (0 5.734))
    (VMON8 DC (0 5.734))
    (VMON9 DC (0 5.734))
    (VMON10 DC (0 5.734))
    (VMON11 DC (0 5.734))
    (VMON12 DC (0 5.734)))
  (outputs
    (OUT0 logical 1)
    (OUT1 logical 1)
    (OUT2 logical 1)
    (OUT3 logical 1)
    (OUT4 logical 1)
    (OUT5 logical 1)
    (OUT6 logical 1)
    (OUT7 logical 1)
    (OUT8 logical 1)
    (OUT9 logical 1)
    (OUT10 logical 1)
    (OUT11 logical 1)
    (OUT12 logical 1)
    (OUT13 logical 1)
    (OUT14 logical 1)
    (OUT15 logical 1)
    (OUT16 logical 1)
    (OUT17 logical 1)
    (OUT18 logical 1)
    (OUT19 logical 1)
    (OUT20 logical 1))
   (off (requires (VCC 0)))
   (powered (requires (VCC (2.8 3.96)) (VCC_IN (2.25 5.5))))
   (configured)
   (monitor value IN1)
   (monitor voltage VMON1)
   (monitor voltage VMON2)
   (monitor voltage VMON3)
   (monitor voltage VMON4)
   (monitor voltage VMON5)
   (monitor voltage VMON6)
```

```
    (monitor voltage VMON7)
    (monitor voltage VMON8)
    (monitor voltage VMON9)
    (monitor voltage VMON10)
    (monitor voltage VMON11)
    (monitor voltage VMON12)
    (alerts voltage VMON1)
    (alerts voltage VMON2)
    (alerts voltage VMON3)
    (alerts voltage VMON4)
    (alerts voltage VMON5)
    (alerts voltage VMON6)
    (alerts voltage VMON7)
    (alerts voltage VMON8)
    (alerts voltage VMON9)
    (alerts voltage VMON10)
    (alerts voltage VMON11)
    (alerts voltage VMON12))



(controller BMC
  (configured)
  (outputs
    (B_PSUP_ON logical 1)
    (C_RESET_N logical 1)
    (PLL_DC_OK logical 1)
    (B_SPI_SEL_N logical 1)))



(controller INA226
  (input VS DC (0 6))
  (input BUS DC (-0.3 36))
  (off
    (requires (VS (0 2.5))))
  (powered
    (requires (VS (2.7 5.5))))
  (configured)
  (monitor current BUS)
  (alerts current BUS))
```

```
(consumer FPGA
  (input CLK clock (0 50))
  (input VCCO_2V5_DDR24 DC (0 3.4))
  (input VCCO_2V5_DDR13 DC (0 3.4))
  (input VCCO_VCC_DDR24 DC (0 3.4))
  (input VCCO_VTT_DDR13 DC (0 2))
  (input VCCO_VTT_DDR24 DC (0 2))
  (input VCCO_VCC_DDR13 DC (0 3.4))
  (inputs
    (VADJ_1V8 DC (0 2))
    (MGTVCCAUX_L DC (0 1.9))
    (MGTVCCAUX_R DC (0 1.9))
    (VCCO_1V8 DC (0 2))
    (VCCINT DC (0 1))
    (MGTAVCC DC (0 1))
    (MGTAVTT DC (0 1.3))
    (VCCINT_IO DC (0 1))
    (VCCAUX DC (0 2)))
  (state OFF
    (requires
       (CLK 0)
       (VCCINT 0)
       (VCCINT_IO 0)
       (VCCAUX 0)
       (VCCO_1V8 0)
       (VADJ_1V8 0)
       (VCCO_2V5_DDR24 0)
       (VCCO_2V5_DDR13 0)
       (VCCO_VCC_DDR24 0)
       (VCCO_VCC_DDR13 0)
       (VCCO_VTT_DDR13 0)
       (VCCO_VTT_DDR24 0)
       (MGTAVTT 0)
       (MGTVCCAUX_L 0)
       (MGTVCCAUX_R 0)
       (MGTAVCC 0)))
  (state ON
    (requires
      (CLK 50)
      (VCCINT (0.873 0.927))
      (VCCINT_IO (0.873 0.927))
      (VCCAUX (1.746 1.854))
      (VCCO_1V8 (1.746 1.854))
      (VADJ_1V8 (1.746 1.854))
      (VCCO_2V5_DDR24 (2.4 2.6))
      (VCCO_2V5_DDR13 (2.4 2.6))
      (VCCO_VCC_DDR24 (1.14 1.26))
      (VCCO_VCC_DDR13 (1.14 1.26))
      (VCCO_VTT_DDR13 (0.57 0.63))
```

```
    (VCCO_VTT_DDR24 (0.57 0.63))
    (MGTAVTT (1.164 1.236))
    (MGTVCCAUX_L (1.746 1.854))
    (MGTVCCAUX_R (1.746 1.854))
    (MGTAVCC (0.873 0.927))))
(sequence (OFF ON)
  (enable CLK)
  (enable VCCINT)
  (enable VCCINT_IO)
  (enable VCCAUX)
  (enable VCCO_1V8 VADJ_1V8
          VCCO_2V5_DDR13 VCCO_VCC_DDR13 VCCO_VTT_DDR13
          VCCO_2V5_DDR24 VCCO_VCC_DDR24 VCCO_VTT_DDR24)
  (enable MGTAVCC)
  (enable MGTAVTT)
  (enable MGTVCCAUX_L MGTVCCAUX_R))
(sequence (ON OFF)
  (disable MGTVCCAUX_L MGTVCCAUX_R)
  (disable MGTAVTT)
  (disable MGTAVCC)
  (disable VCCO_1V8 VADJ_1V8
           VCCO_2V5_DDR13 VCCO_VCC_DDR13 VCCO_VTT_DDR13
           VCCO_2V5_DDR24 VCCO_VCC_DDR24 VCCO_VTT_DDR24)
  (disable VCCAUX)
  (disable VCCINT_IO)
  (disable VCCINT)
  (disable CLK)))
```

```
(consumer ThunderX
  (inputs
    (PLL_DC_OK logical 1)
    (CHIP_RESET_L logical 1)
    (B_SPI_SEL_N logical 1)
    (PLL_REF_CLK clock (0 50))
    (VDD DC (0 1.21))
    (VDD_09 DC (0 0.945))
    (VDD_15 DC (0 1.65))
    (VDD_DDR13 DC (0 1.4))
    (VDD_DDR24 DC (0 1.4))
    ; based on VPP AMRs in JESD79-4B
    (VDD_2V5_DDR13 DC (0 3.0))
    (VDD_2V5_DDR24 DC (0 3.0))
    (VTT_DDR24 DC (0 0.7))
    (VTT_DDR13 DC (0 0.7))
    (VDD_IO33 DC (0 3.6)))
  (state OFF
    (requires
      (PLL_DC_OK 0)
      (CHIP_RESET_L 0)
      (B_SPI_SEL_N 0)
      (PLL_REF_CLK 0)
      (VDD 0)
      (VDD_09 0)
      (VDD_15 0)
      (VDD_DDR13 0)
      (VDD_2V5_DDR13 0)
      (VDD_DDR24 0)
      (VDD_2V5_DDR24 0)
      (VTT_DDR24 0)
      (VTT_DDR13 0)
      (VDD_IO33 0)))
  (state RESET
    (requires
      (PLL_DC_OK 1)
      (CHIP_RESET_L 0)
      (B_SPI_SEL_N 1)
      (PLL_REF_CLK 50)
      (VDD (0.94 0.98))
      (VDD_09 (0.87 0.93))
      (VDD_15 (1.45 1.55))
      (VDD_2V5_DDR13 (2.375 2.625))
      (VDD_DDR13 (1.16 1.26))
      (VDD_DDR24 (1.16 1.26))
      (VDD_2V5_DDR24 (2.375 2.625))
      (VTT_DDR24 (0.58 0.63))
      (VTT_DDR13 (0.58 0.63))
      (VDD_IO33 (3.14 3.46))))
```

```
(state ON
  (requires
    (PLL_DC_OK 1)
    (CHIP_RESET_L 1)
    (B_SPI_SEL_N 1)
    (PLL_REF_CLK 50)
    (VDD (0.94 0.98))
    (VDD_09 (0.87 0.93))
    (VDD_15 (1.45 1.55))
    (VDD_2V5_DDR13 (2.375 2.625))
    (VDD_DDR13 (1.16 1.26))
    (VDD_DDR24 (1.16 1.26))
    (VDD_2V5_DDR24 (2.375 2.625))
    (VTT_DDR24 (0.58 0.63))
    (VTT_DDR13 (0.58 0.63))
    (VDD_IO33 (3.14 3.46))))
(sequence (OFF RESET)
  (enable VDD_IO33 PLL_REF_CLK)
  (enable VDD)
  (enable VDD_09 VDD_15)
  (enable
    VDD_DDR13
    VDD_2V5_DDR13
    VDD_DDR24
    VDD_2V5_DDR24
    VTT_DDR24
    VTT_DDR13)
  (wait 3 ms)
  (enable B_SPI_SEL_N)
  (enable PLL_DC_OK))
(sequence (RESET ON) (enable CHIP_RESET_L))
(sequence (RESET OFF)
  (disable PLL_DC_OK)
  (disable B_SPI_SEL_N)
  (disable
    PLL_REF_CLK
    VDD_IO33
    VDD
    VDD_09
    VDD_15
    VDD_DDR13
    VDD_2V5_DDR13
    VDD_DDR24
    VDD_2V5_DDR24
    VTT_DDR24
    VTT_DDR13))
 (sequence (ON RESET) (disable CHIP_RESET_L)))
```

```
(platform enzian

  #:default-output-accuracy (+- (min (DC 0.02) (% 1)))

  (ISPPAC pac-cpu)
  (ISPPAC pac-fpga)

  (INA226 ina226-ddr-fpga-24)
  (INA226 ina226-ddr-fpga-13)
  (INA226 ina226-ddr-cpu-13)
  (INA226 ina226-ddr-cpu-24)

  (MAX15301 max15301-vcc1v8-fpga 1.8)
  (MAX15301 max15301-util-3v3 3.3)
  (MAX15301 max15301-1v5-vdd-oct 1.5)
  (MAX15301 max15301-vadj-1v8 1.8)
  (MAX15301 max15301-vccintio-bram-fpga 0.9)

  (PSU psu-cpu0)
  (PSU psu-cpu1)

  (Main-PSU main-psu)

  (IR3581-parent ir3581-parent 4.5)
  (IR3581-loop ir3581-loop-vdd-core 4.5)
  (IR3581-loop ir3581-loop-0v9-vdd-oct 4.5)
  (wire ir3581-internal-loop-enable
    (ir3581-parent loop-enable)
    ((ir3581-loop-vdd-core loop-enable)
     (ir3581-loop-0v9-vdd-oct loop-enable)))

  (ISL isl-vdd-ddrcpu13)
  (ISL isl-vdd-ddrcpu24)
  (ISL isl-vdd-ddrfpga13)
  (ISL isl-vdd-ddrfpga24)

  (MAX20751 max20751-mgtavcc-fpga 0.9)
  (MAX20751 max20751-mgtavtt-fpga 1.2)
  (MAX20751 max20751-vccint-fpga 0.9)

  (MAX8869 max8869-mgtvccaux-l 1.8)
  (MAX8869 max8869-mgtvccaux-r 1.8)

  (MAX15053 max15053-sys-1v8 1.8)
  (MAX15053 max15053-sys-2v5-13 2.5)
  (MAX15053 max15053-sys-2v5-24 2.5)
  (MAX15053 max15053-2v5-cpu13 2.5)
  (MAX15053 max15053-2v5-cpu24 2.5)
```

```
(FPGA fpga)

(ThunderX cpu)

(NCP U24)
(NCP U25)
(NCP U39)
(NCP U40)

(SI5395 si5395-clk-main)
(SI5395 si5395-clk-cpu)
(SI5395 si5395-clk-fpga)
(Oscillator oscillator)

(BMC bmc)

(wire b-spi-sel-n (bmc B_SPI_SEL_N) ((cpu B_SPI_SEL_N)))

(wire b-psup-on (bmc B_PSUP_ON)
  ((psu-cpu0 EN) (psu-cpu1 EN) (main-psu EN)))

(wire 3v3-psup (main-psu V33_PSU)
  ((pac-cpu VMON3)
   (ina226-ddr-cpu-13 VS)
   (ina226-ddr-cpu-24 VS)
   (max15053-2v5-cpu13 V_IN)
   (max15053-2v5-cpu24 V_IN)
   (cpu VDD_IO33)
   (ir3581-parent VCC)
   (ir3581-loop-vdd-core VCC)
   (ir3581-loop-0v9-vdd-oct VCC)
   (si5395-clk-main VDD)
   (si5395-clk-fpga VDD)
   (si5395-clk-cpu VDD)
   (oscillator VDD)))

(wire 5v-psup (main-psu V5_PSU)
  ((pac-fpga VMON2)
   (pac-cpu VMON2)
   (isl-vdd-ddrcpu13 VCC)
   (isl-vdd-ddrcpu24 VCC)
   (isl-vdd-ddrfpga13 VCC)
   (isl-vdd-ddrfpga24 VCC)))

(wire 5vsb-psup (main-psu V5SB_PSU)
  ((pac-cpu VCC_IN)
   (pac-fpga VCC_IN)))

(wire bmc-vcc-3v3 (main-psu BMC_VCC_3V3)
  ((pac-cpu VCC) (pac-fpga VCC)))
```

```
(wire 12v-cpu0-psup (psu-cpu0 OUT)
  ((pac-cpu VMON1)
   (max15301-1v5-vdd-oct PWR)
   (isl-vdd-ddrcpu13 EN_VTT)
   (isl-vdd-ddrcpu24 EN_VTT)
   (ir3581-loop-vdd-core VIN)
   (ir3581-loop-0v9-vdd-oct VIN)
   (ir3581-parent VIN)))

(wire 12v-cpu1-psup (psu-cpu1 OUT)
  ((pac-fpga VMON1)
   (max20751-mgtavcc-fpga VDDH)
   (max20751-mgtavtt-fpga VDDH)
   (max20751-vccint-fpga VDDH)
   (max15301-vcc1v8-fpga PWR)
   (max15301-util-3v3 PWR)
   (max15301-vadj-1v8 PWR)
   (max15301-vccintio-bram-fpga PWR)
   (isl-vdd-ddrfpga13 EN_VTT)
   (isl-vdd-ddrfpga24 EN_VTT)))

(wire en-vcc1v8-fpga (pac-fpga OUT15)
  ((max15301-vcc1v8-fpga EN)))

(wire vcc1v8-fpga (max15301-vcc1v8-fpga V_OUT)
  ((pac-fpga VMON11)
   (fpga VCCAUX)))

(wire en-util-3v3 (pac-fpga OUT6) ((max15301-util-3v3 EN)))

(wire util-3v3 (max15301-util-3v3 V_OUT)
  ((pac-fpga VMON3)
   (ina226-ddr-fpga-24 VS)
   (ina226-ddr-fpga-13 VS)
   (max20751-mgtavcc-fpga VDD33)
   (max20751-mgtavtt-fpga VDD33)
   (max20751-vccint-fpga VDD33)
   (max8869-mgtvccaux-l V_IN)
   (max8869-mgtvccaux-r V_IN)
   (max15053-sys-1v8 V_IN)
   (max15053-sys-2v5-13 V_IN)
   (max15053-sys-2v5-24 V_IN)))

(wire psup-pgood (main-psu PGOOD) ((pac-cpu IN1)))

(wire en-mgtavtt-fpga (pac-fpga OUT14)
  ((max20751-mgtavtt-fpga VR_ON)))
(wire en-mgtavcc-fpga (pac-fpga OUT10)
  ((max20751-mgtavcc-fpga VR_ON)))
```

```
(wire en-vccint-fpga (pac-fpga OUT9)
  ((max20751-vccint-fpga VR_ON)))

(wire mgtavtt-fpga (max20751-mgtavtt-fpga V_OUT)
  ((fpga MGTAVTT)))
(wire mgtavcc-fpga (max20751-mgtavcc-fpga V_OUT)
  ((pac-fpga VMON7) (fpga MGTAVCC)))
(wire vccint-fpga (max20751-vccint-fpga V_OUT)
  ((pac-fpga VMON6) (fpga VCCINT)))

(wire en-sys-1v8 (pac-fpga OUT16) ((max15053-sys-1v8 V_EN)))
(wire en-sys-2v5-13 (pac-fpga OUT7) ((max15053-sys-2v5-13 V_EN)))
(wire en-sys-2v5-24 (pac-fpga OUT8) ((max15053-sys-2v5-24 V_EN)))

(wire sys-1v8 (max15053-sys-1v8 V_OUT)
  ((pac-fpga VMON12) (fpga VCCO_1V8)))
(wire sys-2v5-13 (max15053-sys-2v5-13 V_OUT)
  ((pac-fpga VMON4) (U39 VCC) (fpga VCCO_2V5_DDR13)))
(wire sys-2v5-24 (max15053-sys-2v5-24 V_OUT)
  ((pac-fpga VMON5) (U40 VCC) (fpga VCCO_2V5_DDR24)))

(wire vtt-ddrfpga13 (U39 VREF) ((fpga VCCO_VTT_DDR13)))
(wire vtt-ddrfpga24 (U40 VREF) ((fpga VCCO_VTT_DDR24)))

(wire clk-sig (oscillator CLK) ((si5395-clk-main CLK_IN)))
(wire clk-main (si5395-clk-main CLK)
  ((si5395-clk-cpu CLK_IN) (si5395-clk-fpga CLK_IN)))

(wire fpga-clk (si5395-clk-fpga CLK) ((fpga CLK)))

(wire en-mgtvccaux-l (pac-fpga OUT11)
  ((max8869-mgtvccaux-l SHDN)))
(wire en-mgtvccaux-r (pac-fpga OUT12)
  ((max8869-mgtvccaux-r SHDN)))

(wire mgtvccaux-l (max8869-mgtvccaux-l V_OUT)
  ((pac-fpga VMON8) (fpga MGTVCCAUX_L)))
(wire mgtvccaux-r (max8869-mgtvccaux-r V_OUT)
  ((pac-fpga VMON9) (fpga MGTVCCAUX_R)))

(wire en-vadj-1v8-fpga (pac-fpga OUT17)
  ((max15301-vadj-1v8 EN)))
(wire en-vccintio-bram-fpga (pac-fpga OUT13)
  ((max15301-vccintio-bram-fpga EN)))

(wire vadj-1v8-fpga (max15301-vadj-1v8 V_OUT)
  ((fpga VADJ_1V8)))
(wire vccintio-bram-fpga (max15301-vccintio-bram-fpga V_OUT)
  ((pac-fpga VMON10) (fpga VCCINT_IO)))
```

```
(wire en-vdd-ddrfpga13 (pac-fpga OUT18)
  ((isl-vdd-ddrfpga13 EN_PWR)))
(wire en-vdd-ddrfpga24 (pac-fpga OUT19)
  ((isl-vdd-ddrfpga24 EN_PWR)))

(wire vdd-ddrfpga13 (isl-vdd-ddrfpga13 VOUT)
  ((ina226-ddr-fpga-13 BUS) (fpga VCCO_VCC_DDR13) (U39 VRI)))
(wire vdd-ddrfpga24 (isl-vdd-ddrfpga24 VOUT)
  ((ina226-ddr-fpga-24 BUS) (fpga VCCO_VCC_DDR24) (U40 VRI)))

(wire c-reset-n (bmc C_RESET_N) ((cpu CHIP_RESET_L)))
(wire pll-dc-ok (bmc PLL_DC_OK) ((cpu PLL_DC_OK)))

(wire en-vdd-ddrcpu13 (pac-cpu OUT11) ((isl-vdd-ddrcpu13 EN_PWR)))
(wire en-vdd-ddrcpu24 (pac-cpu OUT12) ((isl-vdd-ddrcpu24 EN_PWR)))

(wire vdd-ddrcpu13 (isl-vdd-ddrcpu13 VOUT)
  ((pac-cpu VMON9)
   (ina226-ddr-cpu-13 BUS)
   (cpu VDD_DDR13) (U24 VRI)))

(wire vdd-ddrcpu24 (isl-vdd-ddrcpu24 VOUT)
  ((pac-cpu VMON10)
   (ina226-ddr-cpu-24 BUS)
   (cpu VDD_DDR24) (U25 VRI)))

(wire vdd-core-en (pac-cpu OUT6) ((ir3581-loop-vdd-core EN)))
(wire vdd-core (ir3581-loop-vdd-core VOUT)
  ((pac-cpu VMON4) (cpu VDD)))

(wire vdd-oct-en-l2 (pac-cpu OUT7)
  ((ir3581-loop-0v9-vdd-oct EN)))
(wire 0v9-vdd-oct (ir3581-loop-0v9-vdd-oct VOUT)
  ((pac-cpu VMON5) (cpu VDD_09)))

(wire en-1v5-vdd-oct (pac-cpu OUT8) ((max15301-1v5-vdd-oct EN)))
(wire 1v5-vdd-oct (max15301-1v5-vdd-oct V_OUT)
  ((pac-cpu VMON6) (cpu VDD_15)))

(wire en-2v5-cpu13 (pac-cpu OUT9) ((max15053-2v5-cpu13 V_EN)))
(wire 2v5-cpu13 (max15053-2v5-cpu13 V_OUT)
  ((pac-cpu VMON7) (U24 VCC) (cpu VDD_2V5_DDR13)))
(wire vtt-ddrcpu13 (U24 VREF) ((pac-cpu VMON11) (cpu VTT_DDR13)))

(wire en-2v5-cpu24 (pac-cpu OUT10) ((max15053-2v5-cpu24 V_EN)))
(wire 2v5-cpu24 (max15053-2v5-cpu24 V_OUT)
  ((pac-cpu VMON8) (U25 VCC) (cpu VDD_2V5_DDR24)))
(wire vtt-ddrcpu24 (U25 VREF) ((pac-cpu VMON12) (cpu VTT_DDR24)))

(wire pll-ref-clk (si5395-clk-cpu CLK) ((cpu PLL_REF_CLK)))
```

```
(platform-state both-on
  (cpu ON)
  (fpga ON)
  ; util 3v3 is sensitive, just set it to 3.3 for now
  (util-3v3 3.3)
  (ina226-ddr-fpga-13 configured)
  (ina226-ddr-fpga-24 configured)
  (ina226-ddr-cpu-13 configured)
  (ina226-ddr-cpu-24 configured))

(sequence power-on-both (initial both-on)
  (minimize (enter ON cpu)
            (enter ON fpga))
  ; make sure ispPACs are running before main PSU is switched
  (constraint pacs-on
    (< ((configure pac-cpu)
        (configure pac-fpga))
       ((enter on main-psu)))))

(sequence power-off-both (both-on initial)
  (minimize (enter OFF cpu) (enter OFF fpga))))
```

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Optimizing Declarative Power Sequencing

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Knüsel	Moritz

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Abtwil AG 22.09.27	Mor Knü

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.