ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 410

Systems Group, Department of Computer Science, ETH Zurich

Declarative Dynamic Power Management

by

Roman Meier

Supervised by

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

March 2022 - September 2022

D INFK

Abstract

Modern computers feature large power networks that are non-trivial to safely control. The same is true for the Enzian resarch computer [1], which has a power network with 37 voltage regulators, plus a CPU and an FPGA, both of which impose complex requirements on the order in which their power and clock inputs may be operated safely.

Initially, the command sequences to control the Enzians power network were written by hand, but this proved tedious and error-prone. Luckily, prior work in the systems group at ETH Zürich [2][3] has already solved the problem of declarative static power management, but how to manage a dynamic platform that can change unexpectedly remained unadressed.

In this thesis, we develop the design for a dynamic power management solution that is able to keep track of the changes the hardware undergoes, react to faults and other undesireable platform states, and generate command sequences online to steer the platform into a new state. Our solution can read a platform description from a declarative specification and is therefore not limited to one pre-defined platform. We also show experimentally that our plan generation mechanisms are fast enough for online usage.

Acknowledgements

I would like to thank my immediate supervisors Daniel Schwyn and Dr. Michael Joseph Giardino for their invaluable support and feedback during these six months, and Prof. Dr. Roscoe for his guidance and the opportunity to work in such an inspiring environment.

Deserving of special thanks is Andrea Lattuada, for his help in clarifying sequencing by partial order.

Last but not least I thank my family and friends for their support and understanding during this occasionally stressful time.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

> Bernard, if the right people don't have power; do you know what happens? The wrong people get it!
>
> Sir Nigel Hawthorne as Sir Humphrey Appleby – Yes, Prime Minister

Responsible for the management of the complex power networks on modern systems are (Base)board Managment Controllers, or BMC for short. On modern server-class computers the BMC is frequently an independent, fully-featured SOC with unrestricted access to the power network, communications with the host system (for dynamic frequency scaling or similar power-related requests), and even its own network interface. The firmware these BMCs are running is usually proprietary, or in the best case an open platform like OpenBMC. In either case, the BMC is almost certainly running a complete operating system and taking input from the network interface, usually through a HTTP-webserver.

Considering the extraordinary level of privileged access the BMC has to both security and safety relevant components of the system, one would expect that BMC firmware is implemented with an equally extraordinary level of care and rigour, and that a potential user of such firmware could verify that it would operate their platform safely. However, this is not currently the case. OpenBMC, for example, is a Linux kernel that is running some DBUS infrastructure to allow python and bash scripts to communicate, and the proprietary firmware cannot even be inspected in almost all cases.

To solve these issues, there is an ongoing effort at the ETH Zürich Systems Group to develop trustworthy BMC firmware within the scope of the development of the Enzian research system[1]. The first step, and the one we are concerned with here, is to design and engineer a configurable power and clock management that ideally provides strong safety and security guarantees.

Schult [2] tackled the problem of the static management of a declaratively specified platform. Their solution is able to generate command sequences that take a platform from one power state into another, assuming the platform is "stacically stable", meaning it does not ever change its state on its own without explicit instructions.

Knüsel [3] investigated the possibility of generating platform sates and command sequences that are optimal for some goal state, like minimal power draw, but also relied on the static stability of the platform.

This thesis is now an attempt to take the next step: managing a statically *un*stable platform, a dynamically changing one; to provide dynamic power management. We begin by discussing a few baseline assumptions and building up a common understanding of core concepts, from the top:

The power components present on modern computers are highly heterogeneous, can inhabit complex state spaces and require intricate command sequences to reach those states. Meanwhile, unsafe operation of these power components can cause issues starting from platform instability all the way to causing damage to sensitive, potentially expensive, components. Damaged components, in turn, negatively affect the reliability of the platform or cause it to fail altogether.

At first, any computer is almost entirely inert, bar some coin-battery powered real-time clocks. Once it is connected to power a bare minimum of components usually powers on automatically, like the BMC which is responsible for, among other things, power management of the system and thus also any escape from this "Partial off" state. A much more pleasing power configuration, especially for a system like Enzian, is for it to be "On". "On" being a shorthand for "Useful". What exactly this means can vary drastically from system to system; for an Enzian a, for a user, maximally flexible "On" state would be if both the CPU & FPGA, their respective DRAM and NIC and other connected peripheries were in their respective "On", "Useful" states. We now have some notion of distinct power states a system can be in.

What is left to do is to find some way to connect the "Partial Off" state to the "On" state. We call the transition between two power states a "power sequence", made up of discrete steps the platform can take.

The trivial solution to the generation of a power sequence between two states is to read the specifications of all the power components on the system, study the power network that actually connects them, and manually, quite likely by trial-and-error, come up with a sequence of steps that take the platform from the source to the target state. This is also exactly how power sequences for the Enzian used to be generated, but this approach has multiple, major drawbacks:

1. Any change to the components on the system requires a manual re-generation of the sequence, including certifying that it actually works.
2. Manual work with complicated specifications is error-prone, and while behavioural and safety guarantess can be made for the sequence, doing so usually involves an impractical amount of effort.

Automated tooling is much better equipped for giving these sort of guarantess for an arbitrary instance taken from a large problem space.

Prior work has, as discussed solved "Static Power and Clock Management", which Schult [2] define like so:

> **Definition 6** (Static Power and Clock Management). Static Power and Clock Management is about managing the stable characteristics of conductors. (Schult [2])

Static management explicitly does not concern itself with any kind of deviatory behaviour from what is assumed to be a stable system state. As such, they are not only unable to deal with hardware faults, but the models developed lack the capability to even express

such dynamic behaviour.

For this thesis, we will effectively overload the term "Dyanmic Power Management", which already has an established meaning in the literature. Schult already does so for us:

> **Definition 7** (Dynamic Power and Clock Management). Dynamic Power and Clock Management handles exceptional platform events. This includes general platform failures as well as volatile platform characteristics reaching critical levels. (Schult [2])

Peeking ahead to Section 2.4 we define dynamic power and clock management like so:

Definition 1.1. *Dynamic power and clock management is the process of managing the dynamic power and clock state of a system. Dynamic power and clock management is primarily concerned with the safety of the system under management. In a secondary capacity dynamic power and clock management attempts to reach and maintain a user-defined stable system state. This entails both managing the platform in a (non-faulty) as well as a degraded, faulty state.*

In this thesis, we investigate avenues for solving, and present one possible solution to solve, the dynamic power and clock management problem using a declarative description of the system under management.

Chapter 2

Problem Discovery & Description

> [...] Hilberry was ready to cut this rope with an axe should something unexpected happen, or in case the automatic safety rods failed.
>
> The first nuclear SCRAM mechanism
> Allardice *et al.* [4] - The First Reactor

This chapter provides a high-level overview of our problem, introduces necessary background and nomenclature to give a .

2.1 Enzian

Enzian is a research computer designed by and under development at the Systems group at ETH Zurich.

Enzian is a hybrid computing platform with a cache-coherent server-class Cavium ThunderX-1 CPU and a best-in-class Xilinx XCVU9P-3 FPGA, both equippable with large amounts of independent DRAM, and 400GbE or 80GbE network links for the FPGA or CPU respectively.

Enzian was designed, unlike most other computer systems out there, not with a specific use case in mind or constrained by arbitrary cost limits but as a research platform that should be as flexible as possible.

In particular, Enzian's design is optimized for *Coverage*, meaning that it can emulate as much of the design space of systems as possible, and *Openness*, meaning that as much of the system is available for modification as possible.[1]

2.2 Board Management Controller

On many modern computing platforms, especially server systems, we find a SOC dedicated to the management of the rest of the system. This SOC is commonly referred to as the Board Management Controller.

The Board Management Controller is usually responsible for bring-up and dynamic management of the system, and usually provides users with more or less sphisticated remote management options ranging from serial shell access to SSH and web-interfaces. To make

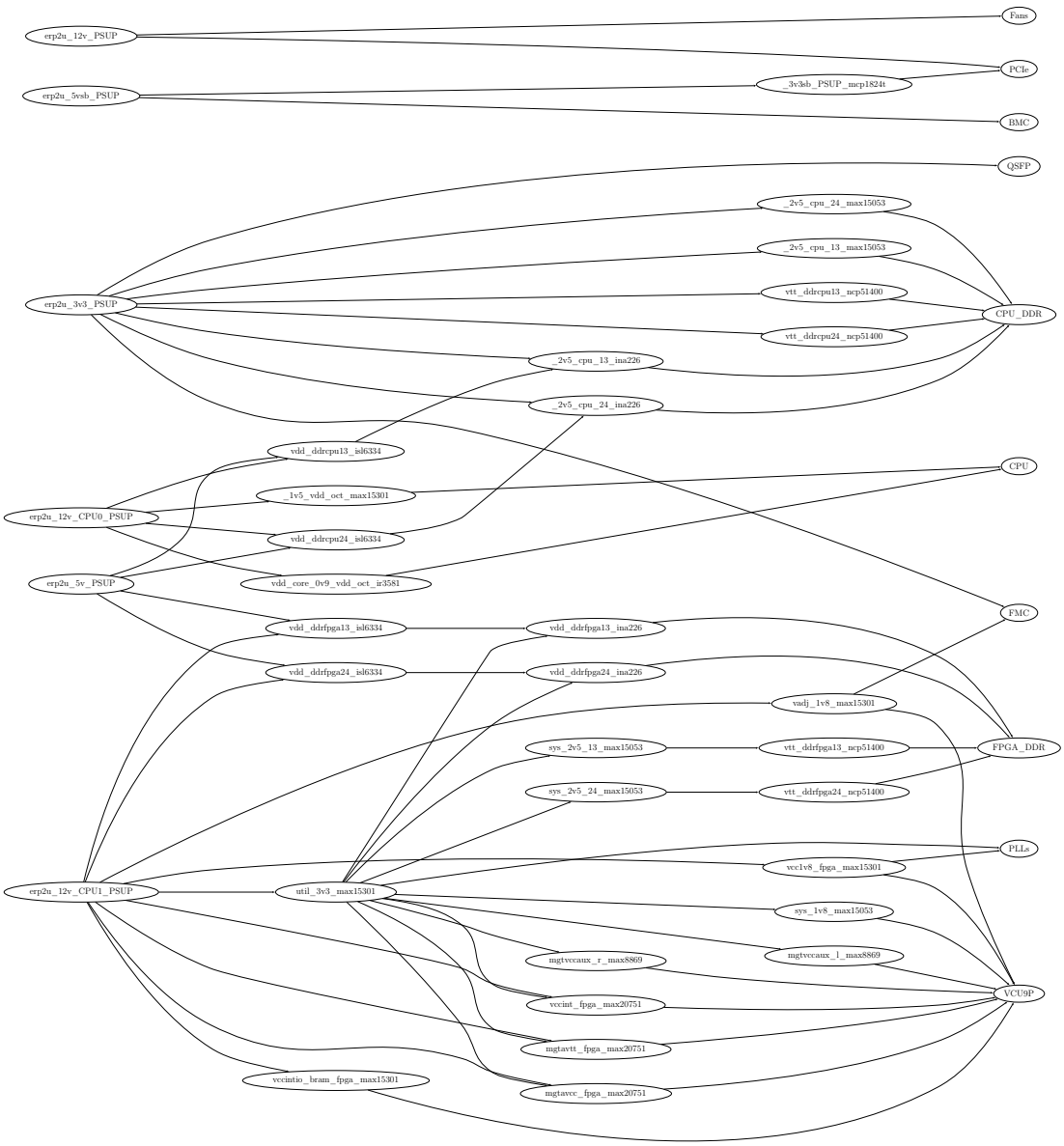


Figure 2.1: A simplified view of the complete Enzian power tree

these remote management capabilities possible the BMC is equipped with a network connection, and many find themselves exposed to the internet directly[5]. Given how critical and potentially vulnerable the BMC is one would expect that it is well-secured and provably so, but it is not uncommon for full Linux distributions to run on these BMCs.

With the development of the Enzian research platform at the Systems group this glaring problem was brought to the attention of the group, leading to efforts to secure the BMC software stack and ideally generate most of the complex, system dependent parts from declarative descriptions.

Initial work has focused on the most immediately critical aspect of BMC funcitonality: power mangement.

2.3 Hardware

We discuss a simple hardware classification scheme as well as how we view the issue of faults & reaction times.

2.3.1 Classification/Terminology

We will avoid using "current" to mean "present", instead opting for "present" directly. We don't expect to talk about gifts, so this is fine.

We follow the classification of hardware into three types introduced by Schult [2] and refined by Knüsel [3]:

Definition 2.1. *Platform A platform is a collection of components and conductors between them.*

Definition 2.2. *Power tree A power tree has the components of a platform as nodes and conductors as edges. For platforms that we are interested in this graph can be represented as a DAG, but in general may be an arbitrary graph.*

Definition 2.3. *Component A component is any node in the power tree.*

Definition 2.4. *Conductor A conductor is an abstract connection between output and an arbitrary number of input ports. A conductor forms an edge in the power tree.*

Definition 2.5. *Producer A producer is a node in the power tree without any power inputs that are independent of the BMC. A producer may still depend on control inputs from the BMC. In the power tree, a producer would be a "root".*

Definition 2.6. *Controller A controller is a node in the power tree that does not drive any conductors itself, but provides monitoring, alerting and configuration pings.*

Definition 2.7. *Consumer A consumer is a node in the power tree without any power ouputs. In the power tree, a producer is a "leaf".*

2.3.2 Faults & Reaction Times

It is in the nature of power faults that if their effects are detectable by sensitive hardware (consumers) then it is already too late.

Ideally, we would like to handle all these faults centrally in the BMC, but because any signal that we might want to send travels at (roughly) the same speed as the fault we have to interpret the fault itself as a signal, which we cannot do as by the time the fault has reached the BMC and gone through the error handling it has surely reached sensitive hardware as well.

Because the only components capable of both detecting and mitigating power faults are the voltage regulators they must be the ones to establish the fundamental dynamic safety of the platform.

Heimhofer [6] measure alert handler run time at $200 - 800$ms for Linux and ≈ 40ms under seL4 and reasons that 40ms is the theoretical limit for an alert handler as implemented due to the fundamental latency induced by I^2C.

2.4 Dynamic Power And Clock Management

A note for the thesis: For brevity, we will frequently only talk about "Dynamic Power Management", but clock management is always implied.

Dyamic power management as found in the literature refers to a completely different problem to the one we are encountering.

While we are concerned with the dynamic management of the systems power network for safety and adaptability, DPM in the literature tries to transition the system into power states such that it consumes as little power as possible overall, as per this definition by Benini *et al.* [7]:

> Dynamic power management is a design methodology aiming at controlling performance and power levels of digital circuits and systems, with the goal of extending the autonomous operation time of battery-powered systems, providing graceful performance degradation when supply energy is limited, and adapting power dissipation to satisfy environmental constraints. (Benini *et al.* [7] – p.xi)

This traditional definition of dynamic power management is unfortunately not useful to us. The Enzian is not battery-powered and is not motivated to reduce power consumption, beyond keeping the FPGA and CPU from frying themselves. DPM solutions in the literature also commonly hone in on how to detect which power state is desirable or optimal, and tend to neglect the actual *control* portion of DPM. We will develop our own definition, starting by cannibalizing the traditional DPM definition for the parts that we agree with:

Partial Definition. *Dynamic power management is the process of managing the dynamic power state of a system.*

Ultimately dynamic power management is not intrinsically motivated. The ideal power state for a system, minimzing all risk, without an external goal is for the platform to be completely powered off. We expand our DPM definition:

Definition 2.8. *The trivial solution to all power managment is for the platform to be as inert as it can be configured to be. We call this minimal state "off" or "powered off".*

Partial Definition. *Dynamic power management is the process of managing the dynamic power state of a system. Dynamic power management tries to reach a user-defined platform state.*

So far our definition basically matches that of static power management. What should intuitively distinguish the two is the idea that DPM is *dynamic*, meaning it reacts to changes in the platform state, be they user-induced or not. This also means that we not only want to *reach* a target state, but also to *maintain* it as far as possible. The decision of when we try to maintain the target state and when we abandon it reveals that we can implement DPM with a goal on the spectrum between

1. Quality of Service
2. Hardware Safety

Remark. *"Hardware safety" in this case refers to physical safety, not logical security. By "hardware that is not safe" we don't mean "logically compromised" but rather "in danger of taking physical damage".*

If DPM tries to maximise *Hardware Safety* then it may be necessary to compromise *Quality of Service* and usage of the system may be impaired. On the other hand, if we maximise *Quality of Service* we may have to risk our hardware taking damage.

Partial Definition. *Dynamic power management is the process of managing the dynamic power state of a system. Dynamic power management tries to reach and maintain a user-defined platform state. A dynamic power management implementation must decide where on the spectrum between maximising "Quality of Service" and maximising "Hardware Safety" its goal lies.*

We argue that for most systems, prioritizing hardware safety makes sense. Only very rarely is hardware considered expendable, and the components we are in danger of damaging are potentially difficult or expensive to replace.

However, as the Enzian is a research system, we do not always want to *maximize* hardware safety either, because that leaves us with a strategy that shuts the platform off completely as soon as anything goes wrong. Instead, we specify that we also want to continue managing the system when it is in a degraded state, and not necessarily always return to the safe "off" state. Part of the motivation behind this is to allow for easier diagnostics when something does eventually go wrong, as a platform that is "off" tells no tales.

Definition 2.9. *Dynamic power and clock management is the process of managing the dynamic power and clock state of a system. Dynamic power and clock management is primarily concerned with the safety of the system under management. In a secondary capacity dynamic power and clock management attempts to reach and maintain a user-defined stable system state. This entails both managing the platform in a (non-faulty) as well as a degraded, faulty state.*

2.5 Distinction from Prior Work

Because there has been prior work in the area of power management, though it concentrated on static management, namely Schult [2] and Knüsel [3], we take a moment to point out some important differences in scope that arise when we can no longer assume static stability of the platform.

2.5.1 Platform State

Prior work did not have to concern itself with the actual state of the platform or how to react to it.

2.5.2 Reactivity

Prior work could assume a completely static view of the world.

This assumption in particular, we argue, has far-reaching consequences: Previously, latency was only a concern insofar as it impacts offline usability of the solution. This means that overall runtime, how it is distributed among solution steps and how runtime varies across executions were far less restricted than they are in our case.

Because we have to deal with actual, dynamically changing hardware we have to make our solution *reactive*, which includes a requirement for low latencies, as we discuss in more detail later on.

2.5.3 Correctness

Prior work - due to it static, offline nature - only had to ensure the correctness of the eventual output of its solutions.

To ensure overall correctness of our approach we also have to, for example, verify the sanity of the declarative platform description.

Prior work took care to never generate a sequence that violated the limits specified in the specification. However, if Prior work were to generate a faulty sequence then the system would simply not turn on. In our case, if we do not generate a correct sequence then the system may cease operating while the user assumes that it is in a stable, safe state.

It is also far less likely that software for dyanmic managment is under human supervision, which is almost certainly the case for its static counterpart.

2.5.4 Optimality

Specifically Knüsel [3] concern themselves with a part of the problem state that we so far have left completely unspecified: Optimality.

We do not pursue optimality as a goal in general, but will comment on it if necessary. While we are much more concerned with keeping the platform safe, solutions with equal capability to ensure safety can be compared on optimality of their operation.

Chapter 3

Solution Requirements

We can now move one step further: developing a set of ground truths shared by all solutions to our problem.

3.1 Bus

We start out by discussing the bus we use to communicate with the components.

We are going to focus in particular on an observation that prior work made that would increase the complexity of our eventual solution: that some unpowered devices can inhibit the proper functioning of the entire bus.

3.1.1 I²C

> SMBus and I²C protocols are basically the same: [...]
>
> NXP Semiconductors - I²C-bus specification and user manual[8]

I²C is a protocol for **I**nter-**IC** (I²C) control and is popular for low-bitrate on-PCB communication. For complete information about the protocol please see the I²C specification [8].

I²C is fundamentally a "master-slave" protocol, meaning there is always one "master" controlling the bus. "Slaves" cannot initiate communication by themselves over an I²C bus, but must always either wait to e contacted by the "master" or use another line of communication, like an alert line as defined in subsection 3.1.2.

The "master" and the "slaves" are connected by a bus consisting of two lines: the serial data line (SDA) and the serial clock line (SCL). SCL is used to synchronize communication between the master and slaves and SDA carries actual data bits, either from "slave" to "master" or from "master" to "slave". The I²C standard defines four bidirectinal modes: "Standard" (100kHz), "Fast" (400kHz), "Fast Plus"(1MHz), and "High-speed" (3.4MHz).

Due to its popularity but generally low-assurance implementations, there has been work on formally specifying and modelling an I²C bus, we refer to Humbel *et al.* [9] for more.

For this thesis, we will end up mostly ignoring the actual I^2C communication and leaving the issue to some external interface.

There is, however, one aspect of I^2C that we are very interested in: the problem with unpowered devices.

Unpowered Devices

Schult [2] first point out that the I^2C specification does not actually *specify* the behaviour of a device that has been powered down, in general. In the worst case, this could mean that an unpowered device pulls the I^2C lines low, potentially disabling the I^2C bus if *any* connected device is turned off.

The I^2C standard is an industry standard that has organically grown over time, we will nonetheless attempt to shed some light on what it has to say on this issue and give our opinion on what it might mean for our expecatations for an "I^2C-compliant" device.

The I^2C standard mentions unpowered devices not being allowed to pull SDA/SCL low exactly thrice. Once in relation to the *optional* "Fast-mode":

> - If the power supply to a Fast-mode device is switched off, the SDA and SCL I/O pins must be floating so that they do not obstruct the bus lines
>
> ([8] – Section 5.1)

And twice in quick succession in relation to another optional feature; software resets after a "general call". A "general call address" is a an address used to talk to all connected devices simultaneously. A "general call address" is followed by a 2-byte sequence, and optionally the standard defines a software reset functionality as follows:

> **0000 0110 (06h): Reset and write programmable part of target address by hardware.** On receiving this 2-byte sequence, all devices designed to respond to the general call address reset and take in the programmable part of their address. *Precautions must be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus.* ([8] – Section 3.1.13 – Emphasis ours)
>
> Following a General Call, (0000 0000), sending 0000 0110 (06h) as the second byte causes a software reset. This feature is optional and not all devices respond to this command. On receiving this 2-byte sequence, all devices designed to respond to the general call address reset and take in the programmable part of their address. *Precautions must be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus.* ([8] – Section 3.1.14 – Emphasis ours)

Neither of these mentions is definitive, however.

For Fast-Mode the question arises of whether this refer to Fast-mode capable devices in *any mode*, or devices *in Fast-mode*.

Consider the following scenario:

> An I^2C bus, connected to which are exclusively Fast-mode devices, all operating in Standard-mode.

Are any of these devices obligated to let SDA/SCL float when they power down?

Consider also the possiblity of Fast-mode Plus devices, which *must* be downward compatible with Fas-mode:

> An I^2C bus, connected to which are exclusively Fast-mode *Plus* devices, all operating in Fast-mode.

Are these *Fast-mode devices* because they are currently in Fast mode? What if they were in Standard mode?

As for the other two mentions, "Precautions must be taken" is possibly the pinnacle of a non-binding statement and it would not be unreasonable to expect hardware designers to implement these precautions with the same level of care as the I^2C standard copy-and-pasted the paragraph into two of its sections.

Overall, we are of the opinion that a *reasonable* I^2C Fast-mode-or-higher device should *definitely* never pull either SDA or SCL low, and that all properly designed I^2C devices in general should be expected to behave the same, unless there is an obvious reason why they should not – if they are designed to be used in a configuration with only one "master" and one "slave", for example.

For this thesis, we will assume well-behaved I^2C devices, so that we do not have to consider a Bus inoperable just because a single connected "slave" is turned off.

3.1.2 SMBus

> Implementations are encouraged to issue a NACK if the [Packet Error Code] is present but not correct.
>
> SMBus specification [10]

The **S**ystem **M**anagement **Bus** or SMBus is an abstraction layer on top of I^2C. SMBus standardises communication over I^2C such that messages with defined semantics can be passed between SMBus-compliant devices. For complete information about the protocol please see the SMBus specification [10].

SMBus was initially designed with the control of "Smart Batteries"

SMBALERT# and the ARA

See appendix A.2 in the standard.

SMBALERT# is an *optional* interrupt line. In theory, a slave device can pull SMBALERT# low to tell the master that it wants to be spoken to. The alert response address (also mentioned in appendix A.2), which the master can use to find out which device pulled the line low, is independent of the SMBALERT# line, so some devices do not react to ARA probes and have to be individually checked for faults when they pull SMBALERT# low.

The standard does specify that

> After receiving an acknowledge (ACK) from the master in response to its address, that device must stop pulling down on the SMBALERT# signal. If the host still sees SMBALERT# low when the message transfer is complete, it knows to read the ARA again. ([10] – Appendix A.2)

which suggests that SMBALERT# and the ARA are intended for event notification and not status signalling.

However, some devices insist on pulling the SMBALERT# line low until their fault condition is resolved, so we cannot simply assume that if the SMBALERT# line is low that there is a new fault, independent of the fault conditions we know to exist on the platform.

Unpowered Devices

As pointed out in subsection 3.1.1 the I^2C specification does not explicitly stop an I^2C device from pulling the data or clock lines low when it is turned off, in general. The SMBus specification has this to say on the matter:

> When the bus is free, both lines are high. The output stages of the devices connected to the bus must have an open drain or open collector in order to perform the wired-AND function as shown in Figure 4. Care should be taken in the design of both the input and output stages of SMBus devices, in order not to load the bus when their power plane is turned off, i.e. powered-down devices must provide no leakage path to ground. (3. - SMBus Specification Version 3.1[10])

This is a considerable improvement over the I^2C specification, but the phrasing of "care should be taken" remains too non-commital to be able to hold any non-compliant device actually accountable.

Two other mentions of the problem suggest a stricter interpretation of the above than we argue is necessarily reasonable:

> The optional diode shown in the diagram above is for ESD protection. It may be necessary in systems where a removable SMBus device such as a Smart Battery is used. However, circuits as actually implemented must comply with the previously stated unpowered leakage current specification. ([10] – Section 4.3.2)

> Because of the relatively low pull-up current, the system designer must ensure that the loading on the bus remains within acceptable limits. Additionally, to prevent bus loading, any devices that remain connected to the active bus while unpowered (that is, their V_{DD} lowered to zero), must also meet the leakage current specification. ([10] – Section 4.3.3)

The second specification still leaves an imporant detail undiscussed: devices that are not "unpowered" ($V_DD = 0V$), but still "off" in some more abstract sense. Those devices are, even if fully compliant with the SMBus specification, still able to pull the bus low and SMBus does not address this issue.

Much like with I^2C, we conclude that we can expect reasonably implemented devices not to pull the bus low if they are not powered or "off", but that the standard again only provides a weak guarantee for this.

3.1.3 PMBus

Plain Text: Characters stored according to ISO/IEC 8859-1:1998 ([A05])

System Management Interface Forum – SMBus specification[10]

The **P**ower **M**anagement **Bus** or PMBus is another abstraction layer on top of SMBus (subsection 3.1.2), defining a protocol specifically concerned with the management of "power converters and a power system". See the PMBus specification [11] for more information.

Unpowered Devices

The PMBus specification includes a clear paragraph to ensure that the PMBus is never compromised by a device "involuntarily":

> As described [in the] SMBus specification, [A03], the electrical signals of a PMBus device must present a high impedance to the bus when the device is unpowered, during startup until fully powered, and during shutdown once the device can no longer assure the proper signal levels. ([10] – Section 4.3)

This precise phrasing addresses all the concerns we had with I^2C and SMBus concerning unpowered devices.

Warnings are an indication of a problem that does not keep the device from operating.

Faults are severe enough that they *may* cause the device to stop operating. Most fault responses are configurable.

A device is allowed to either:

- Set fault condition bits and wait to be polled
- Notify the host of a fault condition

Alerts

For warnings and faults, PMBus devices may implement *one or neither* of the following notification methods:

- Using the SMBus SMBALERT# signal

  PMBus devices pull the SMBALERT# line low if possible.

- Directly communicating with the host

  PMBus devices become temporary bus masters and send notice to the host.

Alert Response Address

PMBus also takes care to clear up any misunderstandings with regards to the Alert Response Address (ARA):

> The SMBALERT# signal remains asserted until is cleared. It is cleared when the device successfully transmits its address in response to receiving the Alert Response Address. It is also cleared by a CLEAR_FAULTS command. ([10] – Section 10.3)

This makes it explicitly non-standard to not release the SMBALERT# signal after receiving a message on the ARA. However, it does not specify that the component cannot simply re-anble the SMBALERT# signal after having it cleared by an ARA or CLEAR_FAULTS command. Persistent faults can still generate multiple signals.

3.2 Timing Requirements & Assumed Stability

To accurately follow the system state we have to consume all events it generates. If events should arrive faster than we can consume them then they must be queued. If they are queued then we either have to be able to determine if incoming alerts/warnings are dependendt on other events in the queue, losing time, or simply ensure that the queue does not grow too big.

As we are dealing with a dynamic system that can rapidly evolve we argue that it is critical that any online DPM solution must be able to *react* to changes in the platform state *quickly*.

If the voltage regulators are unable to ensure the system is in a safe state we must make every attempt at saving the system before it takes damage, but because the kinds of faults that may damage the hardware happen much too quickly for this to be feasible (in fact, they travel at the same speed as any message about them) we have to trust the regulators to keep the platform from taking damage in this way and to inform us about the event.

Thus, if the voltage regulators are always able to ensure the system is in a safe, if degraded, state, then we do not need to immediately worry about the hardware taking damage and this argument for the necessity of fast event resolution disappears.

But even then a secondary concern emerges: that of our DPM solution being able to react *at all* if event resolution does not finish quickly enough.

See Figure 3.1a for an illustration of a DPM having to delay reacting to an alert because a previous event resolution takes too long.

This observation relies on the idea that because events can have unpredictable effects on our model platform state, and that we must keep our model platform in sync with the real hardware to be able decide how to react to most events.

If we assume that some events resolution results are not required for the resolution of some alerts and that the event resolution can in this case be interrupted we run into a new problem:

See Figure 3.1c for an illustration of a DPM livelocking itself because it keeps interrupting event resolution due to further events.

Resolving events faster does not solve either of these issues completely, but it is the only way to lessen their effects and to solve them in most instances.

In conclusion, any DPM solution has to assume that the voltage regulators are able to keep the platform safe independently, and it is our responsibility to configure them correctly to achieve this. Second, we must always consume every piece of information the hardware

sends us, before we make a new decision. This leads to an assumption that this is even possible, i.e. that the hardware does not flood us with events at a rate where we *cannot* keep up with them anymore. We also assume that we always have the time to read the platform state until we are satisfied that we have synchronized our model with it, because the platform would not be observable otherwise, and an unobservable platform is increadibly difficult, if not impossible, to control. Finally, we argue that it is still important for a DPM solution to react quickly to hardware changes, for user experience, and to give our DPM solution a wider range of hardware-event-rates that it can still control.

3.3 Strict Power Dependencies

Dealing with the actual components making up the power network has the unusual consequence that most of the components we would like to model depend on another component to function at all.

Consider voltage regulators A and B with their respective models A_m and B_m. We make no assumptions about how these models are realized, only that they accurately model their targets behaviour and somehow allow their simulation i.e. have predictive capabilities.

If voltage regulator A is connected to voltage regulator B's VCC, then A *must* be in a state where it delivers power to B or cannot transition into a useful state.

This kind of dependence is not actually unusual. The whole point of the modelling techniques we consider as approaches is to describe the situation where one thing happening depends on another thing having happened in the past or not having happened yet.

Consider now the following scenario:

> A is delivering useful power to B, which cannot function without it. B is in an arbitrary internal state and B_m is in sync with B. Voltage regulator A suddenly stops delivering power. As B depends on A for power it immediately turns off.

B_m, to account for B's sudden change of state, *must* immediately reflect this. In fact, B_m must make provisions for B losing power whenever it depends on it, which for any useful device is the case for most of its model space.

This only gets worse as the power network grows in size and more and more components rely on each other for power, depending on the model a component may have to explicitly react to any one of its parents in the power tree changing

For explicitly state-based modelling techniques this lead to an explosion in the number of state transitions that have to be accounted for.

Note that the same thing happens when modelling the complete internal state space of a regulator.

3.4 Hardware Interface Ordering Requirements

Seeing that we can receive distinct categories of events with very different immediate criticality, alerts as "Warnings" or "Faults" or expected measurements, it may seem like a good idea to prioritize alerts over common events.

There is, however, a fundamental necessity in *considering* incoming events in the order that reflects the changes the hardware went through accurately, which for all practical

purposes must mirror the order the hardware produced them in, as we cannot recover this information ourselves in almost all cases.

Removing or disturbing the event order *before* the events have been considered them can lead to the model making incorrect judgements about the state of the platform.

We distinguish between removing ordering information altogether, as happens if we split events and alerts into separate queues, and disturbing the order; swapping events.

There are of course instances where removing or disturbing the ordering is wanted and safe, but those instances explicitly require the prior *consideration* of the events in question.

For most solutions simply consuming events in the order that they are produced by the hardware is ideal.

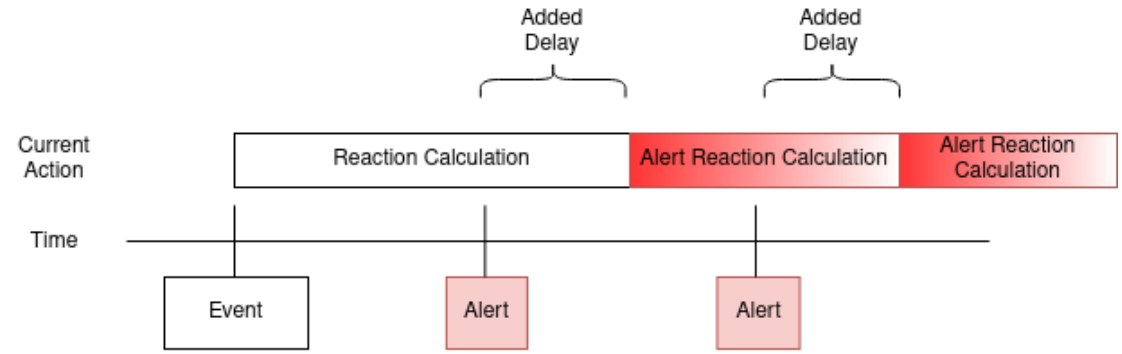
3.5 Infrastructure

3.5.1 Logging

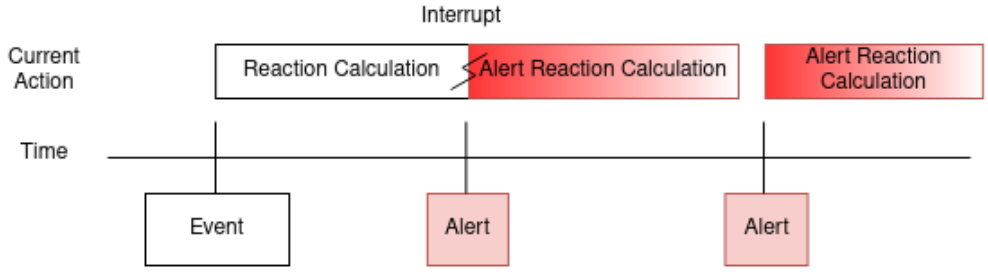
To allow post-incident diagnostics a logging infrastructure is required.

For this purpose we log in two modes: critical and non-critical.

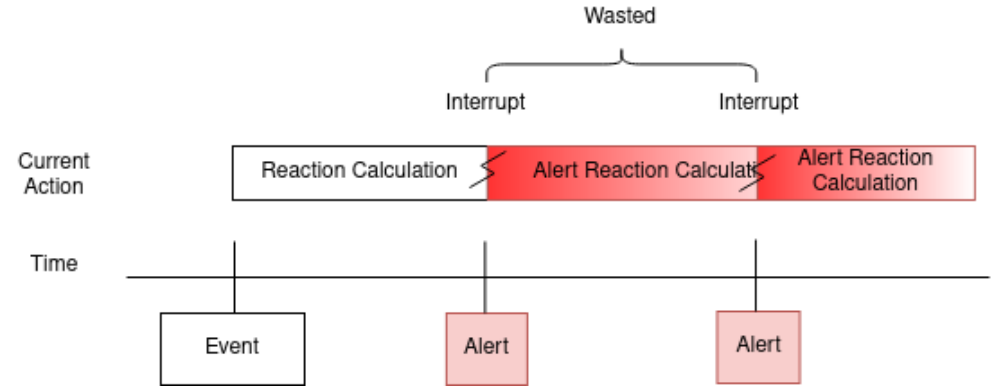
Critical logs must be done instantaneously, no matter what other work is currently being done. Non-critical logging events can be handed off and written asynchronously whenever there is spare time.



(a) Finishing long-running reaction calculations adds delay when new events appear more quickly than they are resolved.



(b) Ideal resolution of Figure 3.1a by interrupting the first event resolution when the alert happens.. Note that the event resolution may not be interruptible in the first place. The system has some time between the first alert resolution and the second alert to react. The system has the chance to avoid the second alert.



(c) Long running alert reaction interrupted by second alert. It is possible that the second alert could have been avoided altogether. Note that the system has not taken a single action and that there is no guarantee that the there is not another alert interrupting us again.

Figure 3.1: Timing requirements explainer

Chapter 4

Approach 1 — Constraint Satisfaction Problem

This approach is effectively what prior work (Knüsel [3], and partially Schult [2], Schult *et al.* [12]) proposed, except that we are also interested in making control decisions at the same time.

For more involved approaches that still may tangentially rely on CSP solvers see the dedicated planning Chapter 5.

4.1 Background

4.1.1 Compilation

Modelling a problem as a CSP is in general quite appealing, but has the major drawback that execution speeds tend to be quite slow.

But there is no need to model a problem as a CSP and then run it from the beginning every time we want to execute it. As an alernative, there has been research into "compiling" CSPs into a format that allows faster solving at runtime, at the obvious cost of flexibility, now only being able to solve the particular CSP that was compiled.

Weigel *et al.* [13] describe compiling partial solutions of constraint satisfaction problems to aide in giving performance guarantees for particular instances while solving them on-line.

Fargier *et al.* [14] provide an overview over different compilation-languages for CSP problems to be compiled into to make solving them online less "adventurous", as they put it.

4.2 Modelling

4.2.1 Sequence Generation

Single-transition sequences

Generating sequences where every conductor may only tansition once given a target platform state is a solved problem, as that is exactly what prior works (partially [2], [3]) already implement.

However, as they both were not conerned with the management of a dynamic platform neither approach is fast enough for online usage on, even comparatively powerful, BMCs.

While both of the prior approaches have some potential for improved runtimes (introducing restrictions from Knüsel [3] into Schult [2], running Knüsel [3] with a simple SMT and not an OMT solver), we are not confident that either could achieve the low latencies necessary for productive online use.

Multi-transition sequences

Prior work assumes that every conductor may only change its state once per sequence to simplify modelling.

This is in part because allowing conductors to change multiple times introducess a variable number of constraints that are solution dependent: the points in time a conductor transitions.

One of the primary reasons we are interested in supporting Multi-transition sequences is that some components can only change their output when their output is low. A reconfiguration sequence for such a controller thus looks like:

$V_{OUT} = xV \Rightarrow V_{OUT} = 0V \Rightarrow V_{OUT} = yV$

and requires the conductor to change state twice.

This sequence is essentially a loop as far as V_OUT is concerned, but there are likely a fair number of unproductive loops that would either have to be explored first, wasting time, or will be part of a generated sequence, as $V_OUT = xV \Rightarrow 0V \Rightarrow yV \Rightarrow 0V \Rightarrow xV \Rightarrow 0V \Rightarrow yV$ is a perfectly valid sequence to generate.

We would thus, in addition, have to consider a "shortest" sequence, further complicating the problem.

As we are already not confident in the single-sequence CSP solving being fast enough, we do not believe this to be a viable approach.

4.3 Custom Strategies

Doubling down on the SMT-solver approach improved solving time can be achieved by implementing a custom strategy for the SMT solver to follow.

Balunović *et al.* [15] for example show that a SMT-solver strategy could potentially be learned from a dataset of formulas, providing significant speedups.

4.4 Summary

While formulating and solving the problem as a CSP makes it possible to solve more general instances of the problem at comparatively little "front-loaded" complexity, the complexity is simply hidden in the solvers used. In general, CSP solvers suffer from poor runtime limits, as sometimes finding a solution can take 0.3 or 10 seconds, depending purely on whether the correct heuristics are chosen.

Especially when trying to not only solve a transition problem with CSPs, but generalizing to include our control problem as well, the problem size would likely exceed what a CSP solver could deal with in a timely manner.

"Compiling" CSP problems has been studied, but we could not glean the runtime guarantees we would have needed ahead of time to justify the time investment into a serious proof-of-concept.

We have thus chosen to not use CSP solvers for online state resolution/sequencing, or the more general control problem.

Chapter 5

Approach 2 — Planning

We discuss planning separately from modelling our problem as a CSP (see Chapter 4), even though some of the proposed solutions for "CSP" are very similar to planning solutions and vice versa. The reason for this is simply that planning has emerged as a separate "field", trying to solve problems very similiar to the one we are trying to solve, while CSP is a much broader term.

5.1 Background

5.1.1 Planning

Schult [2] include a simple planning approach they call "smart backtracking"

In some sense, "Planning" is what prior work by Schult [2] and Knüsel [3] do: taking a platform from some source state to some target state using a set of admissable actions.

> A planning problem traditionally consists of three elements: a description of the planning domain (specifying how actions can affect the state of the world), a description of the initial state of the world, and a description of the goal. A solution to a planning problem is a valid plan—one whose execution is guaranteed to achieve the goal, and whose executability is also guaranteed. (Turner [16], p.1)

We eventually want a solution to a meta-planning problem: How to efficiently find plans, given a model description, between *arbitrary* states. Our initial state is not simply a permutation of a few initial variables, but instead a permutation of the valid states of the entire system. Additionally, *valid* in this case is not equivalent to *safe*, meaning the valid state space is even larger than the safe one.

Generalized Planning

As mentioned earlier, we are not actually interested in finding a sequence between two platform states, but instead sequences between arbitrary platform states.

Generalized planning seems to fit the bill

> Generalized planning is about finding plans that solve collections of planning instances, often infinite collections, rather than single instances. (Bonet *et al.* [17], p.1566)

Jiménez *et al.* [18] give a more elaborate description of generalized planning

> Traditionally the solutions generated by automated planners are tied to a particular planning instance and hence, do not generalize. Generalized planning goes one step further and studies the computation of planning solutions that generalize over a set of planning instances. In the worst case, each instance in the set may require a different solution. In many cases, however, it is possible to compute a single compact solution that exploits some common structure of multiple planning instances. A generalized plan is an algorithm-like solution that is valid for a given set of planning instances. (Jiménez *et al.* [18], p.1)

and strengthen our belief that our arbitrary-sequence problem is essentially a generalized planning problem.

5.2 Modelling

5.2.1 Fault recovery

One functionality that planning could provde for solutions that mainly use other solutions for "good weather" sequencing could be planning fault recovery sequences. In some cases a fault can place the platform in states that it rarely occupies and that a good-weather sequencer may not be able to handle. A planner, on the other hand, if provided sufficient information about the platform, can probably find a sequence if one exists. Deciding when it is safe to attempt the, in all likelihoood more computationally expensive, Planner-fault recovery would prove much more difficult than the modelling of the platform for the planner.

Implementing fault recovery this way would likely require a multi-stage solution with many target platform specific details.

5.2.2 Whole model

An interesting insight is the fact that since it would be possible to make a planner that implmements our minimum interface and the whole problem is essentially a planning task anyway, we could feasibly build a single planner as our solution.

It could be argued that any solution to our problem is a "Planner" in some sense, but here we mean the application of planning-specific modelling and solving techniques to DPM as a whole.

It is unlikely that a tractible implementation could be done, as any naive model of the problem would have a gigantic state space for the planner to sift through.

Without significant simplifications applied to the problem instnaces we expect to solve it is not clear how the state space could be shrunk sufficiently to make such a whole-model planner feasible, and with those restrictions it is questonable whether a planner-sledgehammer is the right tool for the task.

5.3 Summary

We are of the opinion that if the compilation of Planning problems matures or another way is found to ensure execution time limits under all circumstances then Planning could be a great contender for an implementation that does not have to restrict the platform as much as our eventual solution will.

However, for this thesis we decided against using a general "planning" approach. We could either not establish enough confidence that a solution would fulfill the latency requirements, or suspected that we would have to restrict our problem to a degree where another solution becomes more attractive.

Chapter 6

Approach 3 — Discrete Event Systems

In this chapter we discuss the implications of modelling the problem as a single or multiple discrete event systems.

A particular focus will be on Petri nets, as they are a promising method for modelling DES, but we will also discuss discrete event systems as Finite State Automata.

We will ultimately discover that modelling the entire system zealously as a DES is not practical for making large-scale runtime decisions, and, if it were, that the problem domains that DES/Petri nets are applied to in the literature tend to be too far removed from ours to be useful for us. They are well-suited, however, for describing the complet component behaviours that allow us to follow the system state.

6.1 Background

6.1.1 Discrete Event Systems

We begin with a definition of discrete event systems:

> When the state space of a system is naturally described by a discrete set like $\{0, 1, 2, \ldots\}$, and state transitions are only observed at discrete points in time, we associate these state transitions with "events" and talk about a "discrete event system". (Cassandras *et al.* [19], p.26-27)

Why this is an appealing way to model a problem where we have hardware that produces discrete events we can observe, and that we want to model, should be obvious.

Discrete Event Systems, or DES for short, have also been extensively studied, and some useful properties and applications have been identified.

Diagnosis

In the definition of DES, there is mention of transitions being "observable". In real systems, this is not always the case, especially unpredictable events like faults are frequently not observable, at least immediately.

Detecting unobservable transitions, usually after the fact based on the subsequent, observable behaviour of the model is called "Diagnosis".

Diagnosability in DES was first studided by Sampath *et al.* [20] in their seminal paper. They model the DES as a Finite State Machine (FSM) and build another FSM-"Diagnoser" from the system to be diagnosed. The diagnoser FSM receives the same events as the base model, and based on labels on its states decides if there has been an unobservable fault or not. The diagnoser is also able to differentiate between ambiguous and non-ambiguous fault-occurrences.

If we could model a component as a DES, with transitions denoting events that the actual hardware experiences, then we could also build a Diagnoser for that DES, and detect the occurrence of hardware faults that would otherwise be unobservable.

6.1.2 Petri Nets

At their core, Petri nets are descriptions of discrete event systems, like finite state machines.

Instead of only allowing "state" in one place, Petri nets allow multiple places in the Petri net to hold "state" at the same time, vastly increasing their expressive power and potentially allowing for much smaller Petri nets to express a comparatively much larger finite state machine.

For further reading, we recommend Murata [21] for a broad overview with many helpful references for further study and Stremersch [22] for a meticulous, elegant and complete definition of basic Petri nets.

Operational Semantics & Firing Policies

For a list of firing policies we refer to Stremersch [22].

Introduction

Petri Nets were first introduced by Carl Adam Petri in his thesis[23] in 1962. Initially they were not called "Petri" nets, but instead "conflict-free" nets, with their transitions being used for simple bit-manipulations.

Petri nets are more expressive than DFAs, and for most systems more compact.

Unfortunately, that does not mean that they are a good replacement in every scenario, as their analysis also becomes much more complicated.

For larger, more complex systems even Petri nets can reach impractical sizes, most examples used in research tend to be smaller with only few states.

Industry Usage

Due to their versatility and illustrative ability Petri nets have found applications in many fields. They have been used in any field where one could feasibly model the problem as a DES, like business process detection and extraction, esstimating salmon price rises caused by global-warming induced lice[24] or simply for manufacturing systems[25].

Basic Petri Nets

Before we begin we have to note that as Petri nets have been around for a while there exists a great variance in how they are defined and what nomenclature is used to refer to their components. In the following we will use modern terms, as suitable.

A Petri net is a directed bipartite graph with two types of vertices: places and transitions. Directed edges called arcs connect places to transitions and transitions to places.

Murata's definition illustrates this view well[21]:

> **Definition 6.1.** *A Petri net is a 5-tuple,* $PN = (P, T, F, M_0)$ *where:*
>
> $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite set of places,*
> $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite set of transitions,*
> $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
> $W : F \to \mathbb{N}_+$ *is a weight function,*
> $M_0 : P \to \mathbb{N}_0$ *is the initial marking,*
> $P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$.
>
> *A Petri net with the given initial marking is denoted by* (N, M_0).
>
> (Murata [21], p.543)

We note here that the weight function does not change the modelling power of the Petri net [21]. Since F is a set, we could simply replace any a arc with $W(a) = k > 1$ with k many identical arcs.

This graph-based view of defining Petri nets can be deceptive, as it's complexity obfuscates what Petri nets really are: transition-possibilty descriptions given a system state. The definition that best showcases this, and also happens to be one of the most elegant, is given by Stremersch in [22].

n, m are the number of places and transitions respectively.

> **Definition 6.2.** *A Petri net is a triple* $\mathbb{N} = (F^+, F^-, m_0)$ *with* $F^+, F^- \in \mathbb{N}^{n \times m}$ *and* $m_0 \in \mathbb{N}^n$.
>
> where we refer to F^+, F^- as incidence matrices of $\mathbb{N}$. m_0 is the initial state of $\mathbb{N}$. (Stremersch [22], p.8)

This describes a Petri net in its entirety, and all Petri net behaviour can be deduced from it.

We refer to Stremersch [22], p.8 ff for a complete introduction.

Behavioural Properties

This section closely follows the presentation of the same material in Murata [21].

Definition 6.3. *The* Reachability *problem refers to trying to determine if, for a given marking* M_n*, there exists a firing sequence of transitions* $t_i \in T$ *such that* $M_0 t_1 M_1 t_2 \ldots M_n$*. Or in other words: whether a given marking* M_n *is "reachable" from the initial marking* M_0*. Notation:* M_k *is reachable from* M_i *if* $M_k \in R(M_i)$

Definition 6.4. *The* Boundedness *problem is determining if there exists a* $k \in \mathbb{N}$ *such that* $M_i(p) \leq k$ *for all reachable markings* M_i *and all places* p.

Definition 6.5. *The* LIVENESS *problem is deciding if a given Petri net is "live" for all markings M. A marking is "live" if at least one transition is enabled.*

Because liveness is a difficult property to prove it is usually split up into "levels". So a transition t is in any of the four categories:

0. *L0-live/dead* If $t \notin L(M_0)$
1. *L1-live* If $t \in L(M_0)$
2. *L2-live* If $\exists seq \in L(M_0) : |seq, t| >= k$ for some $k \in \mathbb{N}$
3. *L3-live* If $\exists seq \in L(M_0) : |seq, t| = \infty$
4. *L4-live* If t is *L1-live* for each reachable marking $M_i \in R(M_0)$

This liveness-Leveling goes back to Commoner [26] and Lautenbach [27].

Definition 6.6. *A Petri net is "safe" if it is 1-bounded. See Definition 6.4.*

Structural Properties

Definition 6.7. *Controllability A Petri net is completely controllable if any marking of the Petri net is reachable from any other marking.*

Extensions

We have now discussed both basic Petri nets and a few properties these Petri nets can have.

A number of extensions of the basic Petri net model have been proposed. The reason we discuss these separately is because they usually alter both how something is expressed as well as what Petri nets are fundamentally able to express at all.

Inhibitor Arcs An inhibitor arc is a way to express that a state being "enabled" "inhibits" activation of a transition.

An inhibitor arc connects from a place to a transition like a normal arc but cannot connect from a transition to a place.

Given incoming arcs from places $A = \{a_1, \ldots, a_n\}$ and inhibitor arcs from $I = \{i_1, \ldots, i_m\}$, where

$$a_i, i_1 = True \Leftrightarrow a_i, i_1 \text{ is enabled}$$

transition t's truth formula is:

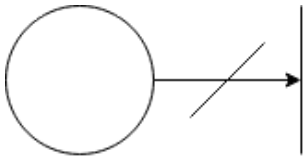
$$t \Leftrightarrow (a_1 \wedge a_2 \wedge \ldots \wedge a_n) \wedge \neg (i_1 \vee i_2 \vee \ldots \vee i_m)$$

Inhibitor arcs were first introduced by Agerwala *et al.* [28] rather trivially like so: given incoming inhibitor arcs from places $B = \{b_1, \ldots, b_n\}$ and normal arcs from $P = \{p_1, \ldots, p_m\}$ to a transition t_1, t_1 is enabled iff:

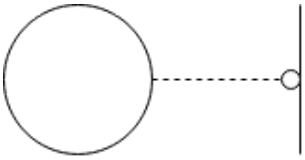
$$(b_1 = 0 \wedge b_2 = 0 \wedge \ldots \wedge b_n = 0) \wedge (p_1 > 0 \wedge p_2 > 0 \wedge \ldots \wedge p_m > 0)$$

Later Agerwala [29] interpret inhibitors like so: they extend a Petri net with a boolean predicate $f : F \to \{T, F\}$ where for an arc $a \in (P \times T)$ $f(a) = T$ means it acts normally and $f(a) = F$ means it acts like an inhibitor.

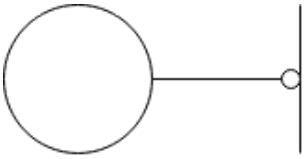
Agerwala *et al.* [28] and [29] draw inhibitor arcs as arcs with strike, Murata [21] draws them as dashed arcs with a circle where they connect to the transition, and in more modern renditions they are usually depicted as normal arcs with circles at the transistion. See Figure 6.1.



(a) Arc as drawn by Agerwala *et al.* [28]



(b) Arc as drawn by Murata [21]



(c) Arc as drawn in modern sources

Figure 6.1: Inhibitor arcs in three styles

Agerwala [29] also showed that Petri nets with inhibitor arcs are equivalent to Turing machines.

To relevant results from this equivalence are:

1. Reachability is undecidable
2. Bounddedness is undecidable

A petri net with inihibitor arcs is referred to as an IPN.

Where 1. follows from the idea that we can build a TM-simulator in an IPN, due to their equivalence. To check if the TM halts we could then solve the *Reachability*-problem for the markings that imply that our TM halts. By this reduction we conclude that *Reachability* is undecidable.

2. follows from the fact that we can, due to their equivalence, build a TM simulator in an IPN. We then trivially extend this simulator with a step counter. k-boundedness of this IPN given a certain TM to simulate then implies that the simulated TM halts after some finite number of steps k, leaving us with the conclusion that *Boundedness* is undecidable by reduction to the halting problem.

Transition Priorities In basic Petri nets all transitions have the same priority, meaning that that all enabled transitions may fire.

Priority Petri nets add a new class of transitions with higher priority; transitions with the same priority behave like normal transitions amongst themselves, but if a higher-priority transition is enabled then no lower priority transitions may fire before it.

Hack [30] proves that inhibitor nets and priority nets are equivalent by providing ways to convert between the two. This also implies that Petri nets with transition probabilities suffer from the same complexity issues as those with inhibitor arcs,

Hack [30] notes that inhibitor nets and priority nets are equivalently expressive. Murata [21] even uses a priority net as an example for inhibitor arcs.

Synthesis

> We understand synthesis as an automated process which, given behavioural specifications or partial specifications of a system to be realized, decides whether the specifications are feasible, and then produces a Petri net realizing them exactly, or if this is not possible produces a Petri net realizing an optimal approximation of the specifications. (Badouel *et al.* [31], p.1)

Petri net synthesis is a fascinating area of research, because it may potentially allow us to generate, or synthesize, behavioural models from voltage regulators, either by simply observing their behaviour, or simply specifying a set of behaviours we'd like the components model to have.

Synthesising Petri nets also has the potential to allow us to give behavioural guarantees for the voltage regulators behaviours.

Theory of Regions

Introduced by Ehrenfeucht and Rozenberg in [32] and [33], the theory of regions is the principle behind much of the synthesis approaches being used. The general idea is to identify "regions" in the graphs given as input, and then usually treat these regions as places in the resulting Petri net.

See Badouel *et al.* [34] for a history on the theory of regions and an overview.

See Ghaffari *et al.* [35] and Lorenz *et al.* [36] for applications.

Fault Diagnosis

The same idea of diagnosis as for FSM-based discrete event systems can of course be applied to Petri nets as well.

We refer to Basile [37] for a good overview of Petri-net fault diagnosis.

Supervision & Control

Given a system described as a Petri net, "controlling" or "supervising" the Petri net is the process of trying to ensure that the Petri net does not violate certain invariants.

In supervisory control, the supervisor, usually also a Petri net, can decide to enable or disable controllable transitions in the supervised net.

Stremersch [22] give a good introduction into supervision:

> The requirement that the system behaviour always satisfies a number of conditions is expressed by a legal set, a subset of the state space. These conditions can express safety, avoidance of deadlock, the language is a subset of a given language, ... Because the controller can only disable events, and thus merely

> limit the set of all possible future paths of the system without being able to 'force' events, one speaks of supervisory control. The supervisory control goal is that the state of the Petri net always belongs to the legal set. Moreover, this needs to be done in an optimal way, namely by keeping the set of possible future evolutions of the Petri net as large as possible. (Stremersch [22], xi)

This view of supervision is useful for systems where the actual state cannot be influenced, say for example signalling control of a train network. The signalling controller cannot actually stop the trains, only enable or disable signals and by extension track sections, for example ensuring that two trains are never in the same section at the same time.

Note that Stremersch call the "supervisor" a "controller". This is a good example of the inconsistent usage of terms by the Petri net literature, and a source of great confusion when initially investigating Petri nets.

6.2 Modelling

Immediately it should be clear why this seems like a promising way to view voltage regulators. Most of the behaviour of voltage regulators that we care about is described either by the voltage regulator being in a certain, discrete, state like POWERGOOD, or by the voltage regulator reacting to some event, like a fault, and changing the state it inhabits, like powering off and stopping power regulation.

Any continuous values we may care about, like output voltage, can be discretized by either binning them into manageably many chunks with a predefined range, like having a state per $0.1V$ of output voltage, or by classifying ranges of continuous values into modal states, like having a state for OUTPUT GOOD, OUTPUT INSUFFICIENT etc. The second approach is, in general, the more natural one as we don't really care about the actual voltage, only about what it means for our system.

Given a complete description of every voltage regulator as a discrete event system we can naturally combine them into a single discrete event system describing our entire system, by having the events from one voltage regulator cause state changes in another.

However, that is where the applicability of discrete event systems seems to come to a halt.

6.2.1 Petri Nets as complete descriptors

Petri nets suffer from having to explicitly specify every transition, which is at its worst when multiple states depend on another one to be active. For every possible transition where the parent state changes there has to be separate transitions for all possible combinations of dependent states that logically exclude each other, as otherwise the system can end up in an inconsistent state where dependent states are still enabled when the parent state is not anymore. In addition, this method requires the circumvention of inhibitor arcs by introducing "boolean places" for each boolean state as we otherwise end up allowing transitions that leave the system in an inconsistent state. There exists the alternative of introducing priority but that makes it more difficult to apply methods developed for general Petri nets, calling into question why we would use Petri nets in the first place.

In addition, Petri nets, while on occasion abused to do so, are not a good way of modelling a single component inhabiting a single state, with complex transitions between possible states. They are instead better suited for modelling systems with states that have simple

transitions, and ideally a place does not denote a "state" of a system, but instead a value or physical object moving through a piece of software or a warehouse.

6.3 Summary

Discrete event systems in general, and Petri nets in particular, are modelling techniques with a lot of history and research behind them.

However, for modelling a power network and controlling it, especially Petri nets are unsuitable, as the size of the state machines, and number of transitions, involved grows far too large to be practical.

A more appealing use-case are simple finite state machines whose only job it is to describe an abstract state of a power-network component, and then to derive control decisions from that.

Chapter 7

Solution

> - Transmuting an `&` to `&mut` is *always* Undefined Behavior.
> - No you can't do it.
> - No you're not special.
>
> Rustonomicon[38]

Having discussed three potential approaches to solving the problem in the previous chapters (See Chapter 4, Chapter 5, Chapter 6) that we concluded were, at least on their own, unsuitable, we now provide our own custom solution in this chapter.

7.1 Background

Most of our solution will be quite idiosyncratic, or referencing prior work from Knüsel [3] or Schult [2]. Some of the background required for our solution was also already discussed in previous chapters.

Still, there are two topics remaining: Partial Orders, which we use extensively for sequence generation, and maximum independent sets, which we use as the basis of our target platform state generation.

7.1.1 Partial Orders

As Schult [2] note, sequencing is about finding a partial order on the events that we want to sequence.

We follow [39] for a definition:

Definition 7.1. *A partial order is established by relation ρ on a set E if*

1. *ρ is transitive in E.*
2. *ρ is irreflexive in E.*

Linear Extension

Interesting for our purposes is the idea of a linear extension:

Definition 7.2. *Given a poset, a linear extension is a total order compatible with the poset.*

because it formalizes our need to realize a total order when we actually follow the partial order as a sequence.

Of course, finding a partial order following our constraints is no use if there is no linear extension. A guarantee that we can always find a total order that also respects the partial order constraints would allow for much more confidence in our solution.

More formally it is required that

Theorem 1. *For each relation π established on a partial order on a set E there exists a relation ρ containing π which establishes a (total) order on E.*

The first published proof of Theorem 1 is Szpilrajn [39].

We can now be certain that, if we find a partial order encoding all sequencing constraints that there exists at least one linear extension and thus an executable sequence for transitioning into the target platform state.

7.1.2 Maximum Independent Set

We use the concise defintion of maximal independent sets from Luby [40]:

Definition 7.3. *A maximal independent set (MIS) in an undirected graph is a maximal collection of vertices I subject to the restriction that no pair of vertices in I are adjacent.*

where *maximal* means that no vertex not in I can be added to the MIS without breaking the IS condition, that no pair of vertices is adjacent.

Definition 7.4. *A maximum independent set $I_{\max}$ is an independent set such that $\forall I : |I| \leq |I_{\max}|$.*

For our purposes *maximum* independent sets are more useful, but we will be using the abbreviation MIS to refer to either depending on context.

Note that $I_{\max}$ is not necessarily unique.

7.1.3 Integer Linear Programming

A *Linear Program* is an optimization problem expressed as a set of linear equations and a linear objective function. For general Linear Programs the variables can take any real value ($x \in \mathbb{R}$), whereas for Integer Linear Programming some or all of them are restricted to integers ($x \in \mathbb{N}$).

While Linear Programming is in $\mathcal{P}$, Integer Linear Programming is $\mathcal{NP}$-complete, and as such in the same complexity class as, for example, solving 3-SAT problems (This fact is rather intuitive, because a 3-SAT problem can be encoded as a set of linear equations with integer variables x restricted to $0 \leq x \leq 1$. Intuitively, the optimization fails if $x \in \mathbb{R}$).

We use the abbreviation ILP to refer to Integer Linear Programming throughout.

7.2 High-Level Overview

In this section we provide a high-level overview of the solution as we envision it.

We provide more in-depth explanations of the solution components introduced here in later sections, and focus instead on the structure of how these components are arranged and their interplay.

We begin our introduction with the most obvious component to consider: the hardware.

See Figure 7.1

Figure 7.1: Initially, we only consider the hardware.

But without having any way to talk to, or interface, with it there's not much we can do, so we define just that: some interface through which we can interact with the hardware.

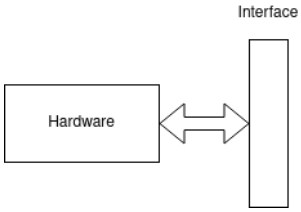


Figure 7.2: The hardware communicates with the Interface.

To allow sending commands to the hardware, which may be inquisitive or imperative, we need some way for the interface to send them *to* the hardware. We also need to receive back information, either about the success of our commands or the results we inquired about.

Because we are concerned with the dynamic behaviour of the platform, we also have to allow for some alerting mechanism from the hardware, whereby it sends us *unprompted* information, and the Interface has to be able to serialize this asynchronous communication for what lies behind it.

On the other side of the interface, of course, sits what we call our "Model".

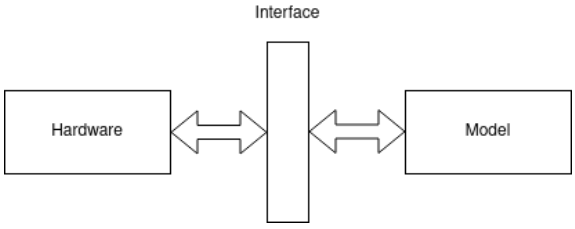


Figure 7.3: Our model uses the Interface to communicate with the Hardware.

Having an abstract interface that translates between the hardware and our model makes sense in our case, because as far as our model should be concerned interaction with the hardware is be homgeneous across devices, even if the devices themselves are not.

The model now ingests information from the hardware, filtered through the interface, and interacts with the hardware when it deems it necessary. The model interacts with the hardware either

- when it needs more information about the hardware state
- when it wants the state of the hardware to change

We argue that any such model that does anything useful must keep some encoding of what it thinks the present state of the platform is. We call this sub-component the "present state"

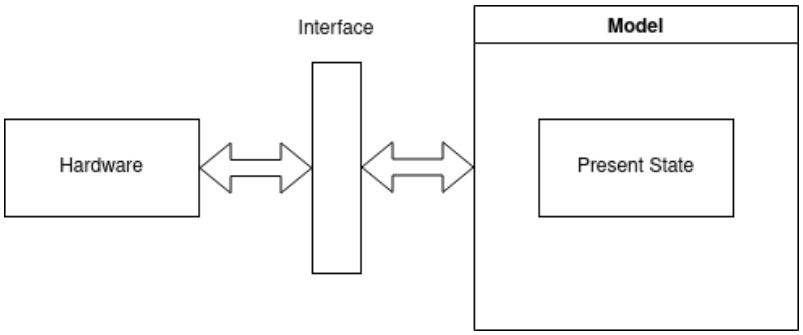


Figure 7.4: The model must keep track of the "present state" of the hardware.

As we will see, this encoding and how to keep it in sync with the hardware is non-trivial, so we split it off from the main model to discuss it separately.

We can finally make the system productive by giving the user a way to interact with the system.

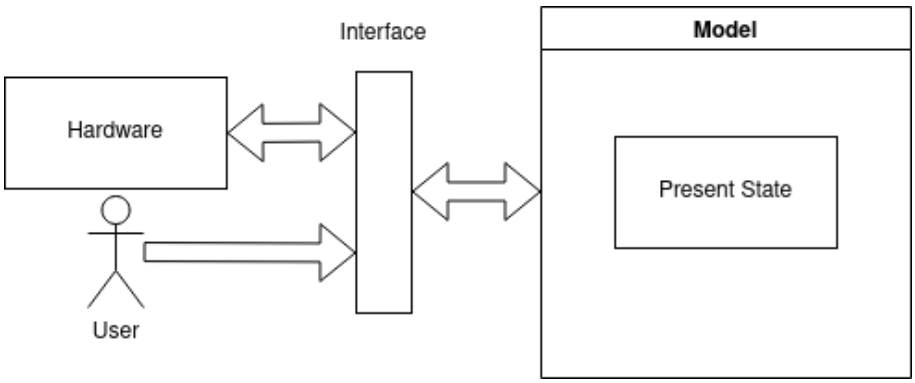


Figure 7.5: Users should also be allowed to send the model requests through the Interface.

The primary interaction a user would have is to have our model *target* some hardware device state, like "turn on the CPU".

Both these interactive user-defined targets and the ones we may want to define to mitigate faults turn out to be non-trivial to reach, as prior work has already shown [2][3].

We therefore introduce the component in our system that will be responsible for coming up with these state transitions, and that our model can query for what to do to reach complex target states.

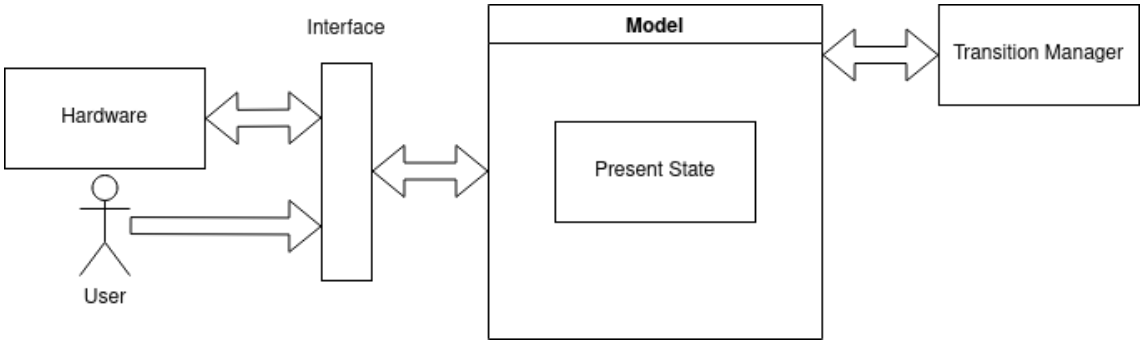


Figure 7.6: A Transition Manager is in charge of ensuring that the platform transitions from one state to another. It keeps track of the transition progress and future transition targets.

By splitting the management of transitions from the "present state" of the platform we are already hinting at another separation that we argue becomes necessary.

While the abstract, static behaviour of power components is generalizeable, as Schult [2] and especially Knüsel [3] have shown, dynamic behaviour is much more idiosyncratic and hetergeneous, some behaviour can even differ between two components on the same PCB which are of the same type, for example due to malfunctioning hardware or user choice.

Conditions like "If this component is starting up and fault A occurs, that is expected and we try again, but fault B causes fault behaviour C" are something that our solution should be able to handle, especially if we want it to be useful in production.

We split our "present state" into two parts: a dynamic "Configuration" component for keeping track of the configuration we applied to the platform dynamically and a per-component abstraction model for determining the present state of components, which will be dynamically updated at runtime but itself static. This also means that we will, and must, handle all fault-detection within these per-component absraction models.

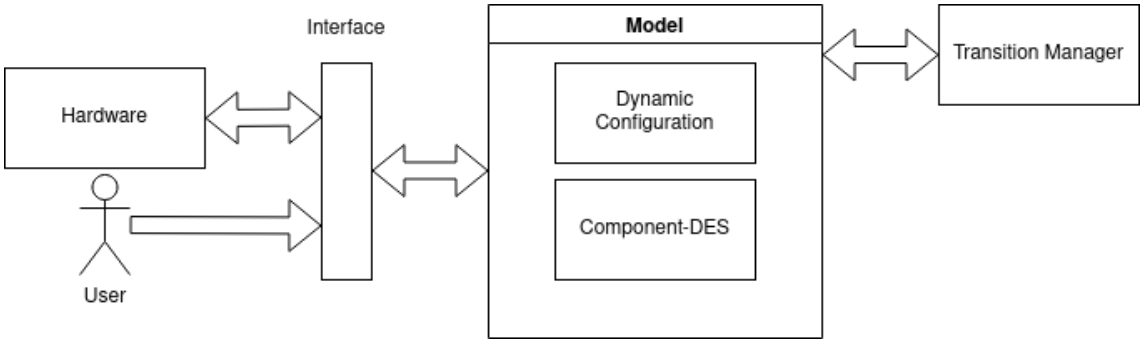


Figure 7.7: Hardware is too hetergeneous, so we model components as DES. Because these DES struggle with keeping much state, we add a configuration management component.

To make it possible to specify these per-component abstraction models externally we decree that they will function as Discrete Event Systems (DES), leave the precise implementation purposefully vague for now, and refer to them as the component-DES. As the name may imply, these component-DES are passed events by the model controller, but also do two additional things: One, the component-DES exposes what it thinks the current component

state is to the rest of the model, and two, informs the rest of the model about actions that significant component changes require, like SCRAMS or sequence invalidations.

7.3 Model Controller

The central component in charge of the model and our interaction with hardware is the model controller.

Because we have to treat the hardware as an entirely asynchronous entity, due to the possibility of alerts, the model controller logic potentially becomes incredibly complicated, having to keep track of an increasing amount of state. For this reason, we introduce a simple model status, derived from the model state, and use this to guide our model controller's decision making.

See Section 7.5 for a description of how we keep track of the present state specifically, especially the component-DES.

7.3.1 Model Status and Model Controller Loop

We observe that we only need three fundamental states to inform our high-level decisions about what the model is/should be doing.

These statuses are:

- Inert. The model is completely satisfied with its current state and without any external impulses (user inputs, sequences to follow or hardware alerts) it would happily remain in this state forever.
- Waiting. Unlike the "Inert" state the model is explicitly waiting for some message from the hardware. The model may respond to a limited selection of user inputs but will not attempt to follow sequences.
- Active. The model considers itself "inconsistent" in some way and wants to *do* something. Currently this either means that the model wants to inquire about some hardware state or command the hardware change.

We call this "model status" i.e. "the model status is Inert" for clarity as "state" is already used elsewhere, primarily for components.

We do not restrict which model status may transition into which model status, or how long the model will remain in or not enter a certain status. Fundamentally, if a model is, for example, never "Inert" then that is perhaps not very useful, because it will never react to user input, but presumably a well-configured model is either doing something else that is more important or the model is actually never *supposed* to take user input, for whatever reason.

This way of modelling platform status is appealing because, given some behavioural guarantees from the functions used, giving liveness guarantees can easily be done simply through the model state.

We could, for example, ask for

$$\Box(\Diamond Inert)$$

which would immediately also provide

$$\Box(step\_available() \Rightarrow \Diamond follow\_step())$$

We can now build up a primary loop for our model controller.

Every loop iteration, we should check for impulses/events that the interface provides for us, deal with them somehow, and then immediately update our model and its status. This should happen independently of the model status, because we may decide to do something in conflict with the platform state, like trying to communicate with a component that just faulted, if we do not ingest platform state information as soon as it is available.

Algorithm 1: Model loop

```
if i ← Interface.impulse_available() then
else
end
update_model();
resolve_new_status();
```

When such an impulse is available, it can take one of two forms: It is either a user-provided target state or the hardware informing us of an update. Uncategorizable inputs are discarded.

Note that any external entity requesting an update is a "User" to us. We do not currently distinguish between a human typing into a terminal and a CPU requesting increased voltage for frequency scaling.

In addition to updating the model we should also give the transition manager a chance to synchronize with the new platform state after updating it. See Section 7.6 for more information on the transition manager.

Algorithm 2: Model loop

```
if i ← Interface.impulse_available() then
    if Impulse i specifies a new user target then
        transitionManager.apply_user_target(i);
    else
        discard(i);
    end
else
end
update_model();
resolve_new_status();
transitionManager.sync_with_model();
```

If the impulse is an update from the hardware, then this update needs to be "announced" to all relevant component-DES. If the impulse concerns a conductor these are all connected components' inputs or if the impulse concerns a component, then that particular component. The DES are then expected to update their state in accordance with this new event. See

subsection 7.5.1 for more information.

Algorithm 3: Model loop

```
if i ← Interface.impulse_available() then
    if Impulse i specifies a new user target then
        transitionManager.apply_user_target(i);
    else if Impulse i is a hardware update then
        announce_to_components(i);
    else
        discard(i);
    end
else
end
update_model();
resolve_new_status();
transitionManager.sync_with_model();
```

Note that, if the Interface provides the model controller with more impulses than it can process, then it effectively deadlocks, never actually executing an action apart from an emergency SCRAM.

This ties in to a fundamental assumption we have to make of the hardware, which is that at some point it, or at least the valid, filtered view of it we see through the Interface, settles down and gives us enough time to process the events it produces. A platform that generates more events than can be processed, while technically observable, becomes uncontrollable because we cannot ever generate a consistent, up-to-date view of the actual hardware state on which to base our control decisions. The set or number of controllable platforms is therefore a function of impulse processing and control decision generation speed, which then provides a ceiling on the number of impulses/events we can handle.

Assuming that the stream of impulses does relent at some point, we finally get to use our model status for deciding what to do.

Algorithm 4: Model loop

```
if i ← Interface.impulse_available() then
    if Impulse i specifies a new user target then
        transitionManager.apply_user_target(i);
    else if Impulse i is a hardware update then
        announce_to_components(i);
    else
        discard(i);
    end
else
    if Inert, and the TransitionManager has a step s it wants to execute then
        a ← action_from_step(s);
        announce_to_components(a);
        execute_action(a);
end
update_model();
resolve_new_status();
transitionManager.sync_with_model();
```

If the model is inert and the transitionManager has some step to execute, i.e. there is a schedule to follow or the transitionManager has targets it can generate a schedule from, then that step is executed.

Previous versions of the solution would attempt complicated "requesting" from the component-DES, but we realized that this is not necessary, as the model status is already Inert and the component-DES should be in a state where they are able to follow the schedule, which they would have cleared if they were not. The drawback of dictating actions to component-DES is that we lose the ability to have component-DES perform additional investigation or actions in preparation to a schedule-instructed action.

See Section 7.6 for more information on how the transition manager handles this, and subsection 7.5.1 for component-DES background.

If the model status is *Waiting* or *Inert*, but there's no step from the transition manager that's available, then the model requires external stimulus to continue operating.

So the model waits for an impulse, and then deals with it in the same way we deal with an impulse when is is not explicitly waiting for one.

Algorithm 5: Model loop

```
if i ← Interface.impulse_available() then
    if Impulse i specifies a new user target then
        transitionManager.apply_user_target(i);
    else if Impulse i is a hardware update then
        announce_to_components(i);
    else
        discard(i);
    end
else
    if Inert, and the TransitionManager has a step s it wants to execute then
        a ← action_from_step(s);
        announce_to_components(a);
        execute_action(a);
    else if Waiting or (Inert without step from TransitionManager) then
        i ←wait_for_impulse();
        if Impulse i specifies a new user target then
            transitionManager.apply_user_target(i);
        else if Impulse i is a hardware update then
            announce_hw_update(i);
        else
            discard(i);
        end
end
update_model();
resolve_new_status();
transitionManager.sync_with_model();
```

And finally, if model's status is *Active*, then the model has to extract the action from the model state – remember that the *Active* model status does not refer to a particular action, but instead the whole model state – and announce to the component-DES that an execution is about to be initiated, before finally executing the action. In most cases, this would mean passing the action off to the Interface, but some Read operations have to be performed locally, as we discuss later in subsection 7.3.2

Algorithm 6: Model loop

```
if i ← Interface.impulse_available() then
    if Impulse i specifies a new user target then
        transitionManager.apply_user_target(i);
    else if Impulse i is a hardware update then
        announce_to_components(i);
    else
        discard(i);
    end
else
    if Inert, and the TransitionManager has a step s it wants to execute then
        a ← action_from_step(s);
        announce_to_components(a);
        execute_action(a);
    else if Waiting or (Inert without step from TransitionManager) then
        i ←wait_for_impulse();
        if Impulse i specifies a new user target then
            transitionManager.apply_user_target(i);
        else if Impulse i is a hardware update then
            announce_hw_update(i);
        else
            discard(i);
        end
    else if Active then
        a ← extract_action();
        announce_to_components(a);
        execute_action(a);
    end
end
update_model();
resolve_new_status();
transitionManager.sync_with_model();
```

Model Status Derivation and PET-Wrappers

Having introduced the model status, what it means for the model to be in specific model statuses, and how we design a controller loop that takes advantage of them the question is now how we derive the abstract model status from more concrete model state.

There is, naturally, an additional requirement for this more concrete state to also encode the actual state of the model such that it is useful for us.

The basic insight that will allow for all this is that we can derive the model status entirely

from the present state of the platform, the state we expect and the state we target.

We collectively call this information, unimaginatively, *MIET* after its four components:

- ***M**easured* state, i.e.
- ***I**nferred* state, which together with the Measured state makes up the ***P**resent* state.
- ***E**xpected* state, the state we expect to be in. We model E as an Option-Enum that can either be $None$, P or T, meaning we either have no expectation of change, expect the value to remain the same, or expect it to change to the target.
- ***T**arget* state, the state we target. Can take any value that *Present* can, with an additional No to encode that we have no target, instead of targeting an unknown $None$ value, which may be reasonable in obscure circumstances.

If we combine the Measured and Inferred state into the Present state we can only consider the *PET* states. See Figure 7.8 for transitions between implied statuses for a single PET value.

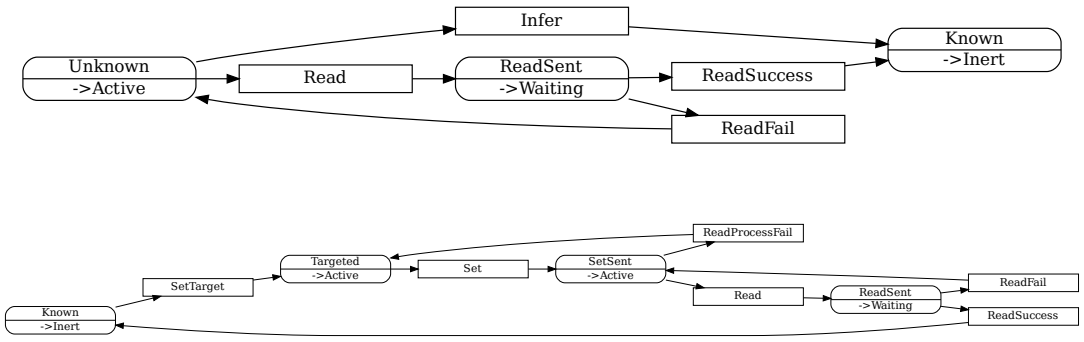


Figure 7.8: Intended PET interaction/actions. States have rounded edges, transitions are squres. Transitions specify the action that must happen for them to fire. States have the model status they imply in their lower half and their name above. "Known" is duplicated and the same state in both figures.

We can use PET as a wrapper for almost all values/states that we want to influence our model status with. In our model we use them to wrap the values of conductors we find interesting (voltage), and whether a particular component has been *configured* or not. Alternatively, a PET can also be used without the ability to produce values, then only deciding model status.

The PET-components also need the ability to encode *None*-values, to encode the absence of any information. For the present component we differentiate between *Unknown* and *None* because the state when we can neither infer nor measure a value is different from when we don't know it yet.

In Rust we can model a `Present<T>` as a `Option<Option<T>>` or as an enum.

To illustrate how to extract the model status from a MIET value we can imagine a function

f that takes P, E, T as its argument and returns a model status:

$$f(p,e,t) = \begin{cases} Inert & \bot \\ Waiting & \bot \\ Active & \bot \\ Invalid & \neg(Inert \vee Waiting \vee Active) \end{cases}$$

We added an Invalid status to catch undefined/impossible combinations. Usually a model would decide to panic and SCRAM if this state is reached without a known fault, though more complicated model repair may be possible.

If we are confident in knowing the present state, are not expecting change and have No target the model is inert.

$$f(p,e,t) = \begin{cases} Inert & p = Some \wedge e = None \wedge t = None \\ Waiting & \bot \\ Active & \bot \\ Invalid & \neg(Inert \vee Waiting \vee Active) \end{cases}$$

If we ever expect the present then we are waiting for information.

$$f(p,e,t) = \begin{cases} Inert & p = Some \wedge e = None \wedge t = None \\ Waiting & e = P \\ Active & \bot \\ Invalid & \neg(Inert \vee Waiting \vee Active) \end{cases}$$

If we have a target and are expecting it then we want to take action. Note that this covers both the cases where we haven't sent the *Set* action yet and when we have sent a *Set* but haven't confirmed it through a read.

$$f(p,e,t) = \begin{cases} Inert & p = Some \wedge e = None \wedge t = None \\ Waiting & e = P \\ Active & e = T \wedge t = Some \\ Invalid & \neg(Inert \vee Waiting \vee Active) \end{cases}$$

In addition, if we don't know the present value we should endeavour to find out:

$$f(p,e,t) = \begin{cases} Inert & p = Some \wedge e = None \wedge t = None \\ Waiting & e = P \\ Active & p = Unknown \vee (e = T \wedge t = Some) \\ Invalid & \neg(Inert \vee Waiting \vee Active) \end{cases}$$

But halt! This way we may be Active when we should be Waiting. We choose to only be active if we wouldn't wait otherwise.

$$f(p,e,t) = \begin{cases} Inert & p = Some \land e = None \land t = None \\ Waiting & e = P \\ Active & \neg Waiting \land (p = Unknown \lor (e = T \land t = Some)) \\ Invalid & \neg(Inert \lor Waiting \lor Active) \end{cases}$$

It is not immediately clear what to do if we do know the present state, are expecting no change but have a target. If we consider it a valid state then it could be either Inert, if we think that not expecting any change is uniquely inert, or Active, if we want there to be a $e = None, e = T, e = P$ sequence whenever we set a target. We choose here to consider it an Inert state. This gives us more fine-grained control over when we want the model to act on a target and allows us to tell the model about future targets that are not active yet. Because implying an Inert state also effectively makes the model ignore the value in this PET-wrapper, this state becomes a "Hint" to the rest of the platform.

$$f(p,e,t) = \begin{cases} Inert & p = Some \land ((e = None \land t = None) \lor (e = None \land t = Some)) \\ Waiting & e = P \\ Active & \neg Waiting \land (p = Unknown \lor (e = T \land t = Some)) \\ Invalid & \neg(Inert \lor Waiting \lor Active) \end{cases}$$

We note here that the PET value is not sufficient to, for example, tell *which* action we should take. All the *Active* status means is that there is *something* to do, but a little bit of additional state is required to differentiate between the case where we have sent a *Set* and when we send a *Read* to confirm.

7.3.2 Model Controller Operation

Having specified the model controller loop, model status, and how to derive it, we must now discuss how exactly the model controller implements the actions from the control loop.

We will skip over discussions of how the model controller communicates with the interface, as that is discussed seperately in Section 7.4.

When the model controller receives a hardware event or executes an action, then that constitutes novel information about the hardware, and the affected component-DES must be informed about its existence.

The rules for which components get informed are rather simple: for events and actions that affect a conductor they are distributed to all components that have an in- or output connected to the conductor. If the event or action instead targets a specific component (anything that involves configuring a component, for example) then only the affected components DES is told about the event.

To extract an action from the model we go through all components and conductors in the topological order of the power-net. For every component and conductor, we compute the

action implied by their PET-wrappers (voltage for the condcutors, the configuration for the components).

Ultimately, we only want to extract a single action, and so we choose them according to this hierarchy:

- Lowest in the power-tree, implying a Wait.
- Lowest in the power-tree, implying a Read. (via the readNext boolean value that we added to every conductor-PET wrapper), or a Write/Configure. Tie-break in favour of Reads.

The highest priority should naturally be to wait for another action to complete, otherwise we could end up in a situation where we are constantly starting new actions. We want to ensure, however, that we only ever have a single request to the hardware outstanding, because one action may override another if we are not careful (Allowing multiple pending Reads, but only a single Write, at any time may be doable, but we do not consider this further).

We then follow the simple logic that whichever action is lowest down the power-tree has the highest potential effect on the rest of the power-tree and should be prioritized.

Whether to favour Reads or Writes is a non-trivial question. If we favour Writing, then we may reduce the amount of time wasted on Reading state when a component wants us to correct a fault. Preferring Reading makes sure that the model always reflects our best possible understanding of the actual hardware state. We compromise by preferring whichever is lowest down the power-tree. This still makes is possible for the power-tree to be well-known as far up as necessary, but required Writing still gets done once the state-defining lower power-tree is resolved.

Once we have extracted an action, we announce the action to the relevant components, as described above. When executing the action, it is sent off to the interface, unless it is a Read action for a conductor value.

While all actions are not always executable, for all actions except Reads this is an automatic Failure condition and the interface can reliably report it back to us as such. For Reads, however, it's possible for a conductor to not be monitored, and thus not be readable, but we still want to be able to "Read" the conductor value for simplicity. What we do in this case, is to "infer" the vaule of the conductor, by looking up the value that is currently assigned to the conductor according to our model and then injecting the result into an impulse.

Note that this is a fair thing to do, because if we are reading the output of a conductor then its inputs are known and the component-DES considers them certain, otherwise we would have extracted and executed the action to ensure that some input is consistent.

Extracting an action from a schedule step, announcing and executing it works in much the same way as it does when extracting one from the model.

7.3.3 Configuration Management

Configuration management involves keeping track of the dynamic values of conductors, inputs and outputs and returning them to whoever requests them.

When a device is configured, in addition to actually executing the action, a copy of the configured in-/output limits and exact assignments are saved in the configuration manager.

When components later want to test the value of an in- or output they know has been configured to a certain value, they, or the controller, query the configuration manager for them.

7.4 Interface

Knüsel [3] reduce their interface to the actions

- Set
- Configure
- Wait
- Monitor

and then translate these calls into python code that can be used as a sequence in the existing Enzian power manager.

We maintain the interface from Knüsel [3], but replace "Monitor" with "Read". The monitor action explicitly waits until a certain monitor has reached some value, which we do implicitly.

Using a PET state (p, e, t) for waiting we can ensure that $e = P$ whenever we wait and reset it to $e = None$ whenever we receive a wait result. If we keep $p = None$ and don't change it, we keep switching between *Inert* and *Waiting*. Because the model status prefers *Waiting* over *Active* we are certain that no two wait actions conflict.

7.4.1 Hardware Interaction

For each command the interface decides how to execute it based on parameters it is provided with.

Setting the BMC output B_PSUP_ON to 1, for example, requires interacting with the GPIO pin of the same name on the BMC.

Reading the value of the monitor $VDDH$ from a *MAX20751*, on the other hand, is a blocking I^2C call the result of which is passed back asynchronously to the model.

7.5 Present State

Before we dive into how we can keep track of the present state of the hardware we have to re-affirm our view that this is a fundamentally hopeless endeavour.

Assuming a supernatural ability to completely freeze the hardware whenever we are deliberating its state we could conceivably build a model that accurately reflects the *abstract state* of the hardware at that particular point in time. Because the full internal state of controllers, and especially consumers, is

- not measureable
- not documented in sufficient detail

we cannot even build a model granular enough to keep track of it.

Re-introducing real-time we find that the hardware, as an independent, dynamic structure can change fundamentally while we are still reacting to the last bit of information we got from it.

Thus, talking about the *present* state we have to add the caveat *as far as we can know it.*

If the complexity here is not appropriately handled and no suitable interface for querying the present state provided, then it can easily seep into, and complicate, every part of a solution that wants to know the state of the platform, which happens to be most of it.

This section describes both the difficulties with determining the present state of the platform and our attempt at dealing with them.

7.5.1 Component-DES

As hinted at when describing our platform model at a higher level, we intend to partially sidestep the complex issue of determining the present state of a component by offloading that responsibility to a user-provided, per-component DES in one form or another.

We will now provide a specification first for what we require the DES to take as input and what we need as outputs, and then a set of "guidelines" and examples for the behaviour the DES should implement to work well in our system.

In- & Outputs

A DES takes as input a single event, which it consumes and uses to change its internal state. Once it has consumed the input event, it may also change what it outputs. A DES may explicitly not change its outputs without an input event.

As a quick overview: a DES for component C receives information from the model controller via events informing it of both the initiation and the completion of hardware changes, when they concern connected conductors or component Cs state directly. The DES then outputs an abstract state representation and transition information that the transition-manager and the rest of the system can work with, as well as request changes to its connected conductors and its configuration status via PET-wrappers.

In more detail, a DES for a component C receives events from the model controller informing it of events in two classes: events concerning a connected conductor and events concerning the component itself. The events concerning a conducer can be further broken down into those connected to a input and those connected to an output. A DES will *only* receive events that relate directly to it or the conductors it connects to, no events concerning any part of the rest of the platform.

Because we expect the DES to keep only minimal state, the DES is informed both via event when a hardware change is initiated as when when it is completed, so that expected and unexpected events can be distinguished. We refer to the these two complementary events as "Initiate" and "Complete" events.

When we specify that an event is *parametrized* we mean that it carries value information, usually a range.

The DES is informed about the following events:

- Read, Infer: Parametrized, initiated hardware events that inform the DES that a particular value or range was read or inferred for a connected input. These events

share an initiation event, which is simply a Read, because for our model we do now know that we will infer a value until we try to read it.

- Alert: A parametrized, uninitiated hardware event. Carries information about the component/input/output affected, for components the type of fault, for conductors the implied value range, and a severity of either FAULT or WARNING.
- Configure, Deconfigure: An unparametrized, initiated hardware event informing the DES that the component has just been configured or deconfigured. The actual values used for configuration are determined outside the DES, which only needs to know that the action has occurred.
- Write: A parametrized, initiated hardware event that gives feedback when a controller output is written, like the GPIO pins on a BMC. Regulator outputs are not written, but have their values set via configuration.
- Wait completion: An initiated event informing the DES that a Wait request this DES had was serviced. Not parametrized, as that would require the DES to keep track of remaining wait time.
- Failed: A wrapper event for any of the other complete event types. This communicates that there was an attempt at Reading or Configuring a conductor or component, but the execution failed. This is separate from a "Fault", because a "Fault" communicates that something happened unprompted, whereas a Failure happens explicitly because of some action we attempted. An initiation event cannot fail.

Optionally, a DES may output commands to inform the rest of the system that something fundamental about the component has changed that is impactful outside of it, or that it requires some service that it cannot be expected to provide itself, for example the relative timing of events. These may be emitted after each event.

- Invalidate Sequence: Must be emitted if either the abstract states this DES outputs change, or the transitions between them.
- Invalidate target platform state cache: Must be emitted if the abstract states this DES outputs change, to inform the transition manager that it has to clear any caches it may keep. If only the transitions between states or the transitionary present state changes, then invalidating the state cache is optional.
- Target state: A component can request a new target state for itself. This target state is push'ed to the transition manager for sequencing and execution. This feature should be used sparingly and only if strictly necessary, for example after a fault or warning.
- SCRAM: Orders the model to immediately SCRAM.
- Wait: Asks the model to halt operation for a specific amount of time, in milliseconds. Useful to enforce component delay requirements.

It is very important to note that, apart from a SCRAM, these can currently only be executed *in addition to* and not *in lieu of* whatever prompted their emission. A requested target state will thus only potentially be targeted much later, for example.

A DES is only useful, of course, if it produces an output, which in our case consists of the following:

- For every in-/output as well as the configuration state of the component, a PET-wrapper.

  For the configuration state this is a PET-wrapper boolean that is expected to carry actual configuration information, i.e. whether the component is configured or not. The configuration-PET must not ask to be read, only written, because configuration information cannot currently be recovered from the platform.

  For the component in-/outputs the PET-wrapper contains either a value/range or a CONFIG marker that implies that the current value for this in-/output should be read from the configuration cache.

- Optionally, a full, reduced Knuselian abstract state (see subsection 7.5.3) set including a a designated state from that set that is the "present state" of the component. This state set must include transition information for all the states.

  Optionally, the present state can be a virtual "transitionary" state that is only used for finding sequences, but not target platform state computation. A transitionary state sits in-between two full states, has the same assignments as the last full state the component was in, and dynamically calculated requirements and transitions to either full state. These transitionary states become necessary if we want to be able to invalidate sequences and then re-calculate them, the alternative would be to return all components to full states whenever we invalidate the current sequence, which may not always be possible and involve tedious manual computation to reverse the already executed sequence, which we just determined to be invalid and couldn't serve as a basis for this new sequence.

Theory of Operation

The component-DES has threefold functions: to expose the current state of the component to the model via its outputs, to force the model to take corrective action if the safety of the component is jeopardized, and to ensure safe operation of the component.

The component-DES does *not* have other control responsibilities, and is, as far as possible, *told* what to do and only makes independent decisions when absolutely necessary. An example of behaviour that does not fall under component-DES responsibilities is, for example, repeated reads until a certain condition is met, if that is required for sequencing. The intended way to solve this would be to have the sequencing explicitly request multiple reads, and for the component-DES to return to the same original state every time. A component-DES would, however, absolutely be allowed, and even expected, to do a "Wait Until" if it is required for the safe operation of the component.

The component-DES is also expclicitly expected to ensure that the components sequencing requirements are observed, for example by only confirming that an in- or output has changed once it has been confirmed by a Read.

That said, the exact reason for using these complicated DES for component control is that we expect commponents to have idiosyncratic, unpredictable behaviours, it is difficult to write a behaviour specification with the same level of strictness as the description of what the in-/outputs of the DES should mean.

Nonetheless, to make it possible for separate DES to work in harmony some behavioural ground-rules are necessary:

1. If a DES ever looses track of the exact component state, suspects it is out-of-sync, or receives an otherwise unexpected event, then it is appropriate for the DES to ask for a SCRAM. Ideally, the DES tries to re-establish a coherent state, but a scenario like this implies either an, at best, unusual system state or a configuration error, both of which should be considered critical.
2. When a DES consumes an Initiate event it should be ready to receive the associated complete event.
3. A DES should start out in a state where it is "inquisitive", i.e. where it does not have a present state and uses its component in-/output-PETs to signal that it wants to read the value of at least its inputs.
4. When a DES advertises a state transition and that transition occurs normally, barring any faults or other events, then the DES must emit a state congruent with the transition having occured. This requirement is vital to the proper functioning of the sequencing.
5. When a DES outputs a target state request that target state must be reachable within the currently emitted states.
6. A DES should try to simultaneously accomodate as many different events potentially occuring as possible. A DES should be prepared to deal with, for example, component in-/outputs being read without the DES requesting it, but also ingest the information gained from such events. This requirement can be important for keeping the diferent DES synchronized.

We are confident that a power-tree of components with DES implementing these requirements will be able to function properly.

Realization

While we want to call attention to the fact that many valid implementations of such a DES are possible, we choose here to view it as a DFA, or discrete finite automata, for two reasons:

- DFAs are still very general and many alternate implementations could be interpreted as being abstracted by a DFA.
- DFAs are well-studied and have great potential for optimization, analysis or specification based on this research.

Another benefit to using a DFA is that its construction is rather simple, and our description the component-DES' interface and capabilities map to DFAs without much further elaboration: a DFA that implements a DES takes on states that emit the outputs the component-DES specifies, and the DFA state transitions take as "symbols" the input events, again as specified by the component-DES.

We unfortunately will have to leave the, very intriguing, further investigation of the encoding and design of these DFAs as future work.

7.5.2 Reading Hardware State

When a *Read* action is executed by the model controller for some conductor c, then the model controller must first check if c is monitored in the present model state. If c is

indeed monitored, i.e. at least one monitor is active that is connected to c, then the model controller sends a *Read* request to the Interface.

If c is not monitored, then we cannot ask the hardware for the actual value of conductor c, and have to fall back on inference.

Inference is a potentially highly complex operation, but we drastically simplify it by means of our component-DES, whose judgement on their outputs we must trust, and the simple assumption that for the whole power tree below c, the present state is known and accurate.

Under this assumption, we infer the value of c to be whatever the component-DES of the output connected to c says it is. For this we check the output-PET the component-DES emits and either extract the value directly or look it up in the dynamic configuration, if the output-PET's value is CONFIG.

This avoids the rather tedious business of inferring values *down* the power tree, but luckily we can convincingly argue that an input's value only changes when the connected output changes, making the output connected to a conductor the principal authority on the conductors value. The component output is itself determined by its inputs. Assuming the output of a conductor changes, but its inputs don't, and this process generates no fault or other event, then the platform is not properly observable and does not follow a fundemantal assumption we have to make about it in order to be able to control it. Generally, whenever two hardware states that differ in their semantics, and that difference is relevant, are indistinguishable from one another, then observability is violated and in most cases controllability as well. But because the occurrence of such a scenario is, by definition, unobservable, we cannot react to it all, including emergency measures like a SCRAM. A platform that does, and one that does not, behave in such unobservable fashion are themselves indistinguishable, and so it is up to us to decide which one of these we assume for a platform. Since the unobservable option is not useful, we pick the observable variant.

Returning to our discussion of inference, we are allowed to assume that the power-tree below c is known if we pick Read actions in bottom-up order relative to the power-net and prioritize them over Writes and Configures.

7.5.3 Restricted Knuselian Component States

To model the present state we mostly follow Knüsel [3], but will introduce additional restrictions that make our dynamic solution more feasible.

Basics

The two major building blocks of the platform state are

- Conductors, or sometimes "wires".
- Components.

We will divide the components into separate classes later. For now we simply assume that they all have

- A unique name
- Ports:
  - Inputs, to which conductors can connect.

  - Outputs, to which conductors can connect.
- Monitors, which inform us that a port's value can be *Read* by querying the component in some way.
- Alerts, which inform us that this component can raise alerts for a certain inputs values.
- Some abstract states, that "require" inputs and "assign" outputs.

Ports also define a safe range of operation. The *Free* assignment makes use of this to reduce some of the tedium of maintaining a specification.

We define the *Ports* to have a *Type*:

- *Logical*, with a restricted subtype *Boolean*
- *DC*

A *Conductor* may connect one output to any (positive) number of inputs, but the output must have the same *Type* as all the inputs it connects to. Indirectly, this also makes the conductor type the type of the output they're connected to.

A component state is a named set of distincg input requirements and output assignments, that includes transition descriptions to other component states.

Partially following Knüsel [3], we allow Assignments in one of three forms:

- Value assignments assign a static value to an output
- Value-Range assignments assign from a static range of value
- Free assignments are shorthands for Value-Range assignments, with the range being the whole safe range of the output.
- InputMatch assignments, which specify an input to match and assign to the output the value of the input. InputMatch assignments significantly complicate state generation and sequencing, but are necessary for modelling the Enzian, which includes clock multipliers that pass through input frequencies with an optional multiplier.

Requirements are much simpler than assignments. A requirement simply refers to a component input, and includes minimum and maximum values that the input may take.

Restrictions

Our state so far allows components to inhabit and change into arbitrary states. This will be significantly restricted in this section, firstly to make declaratively specifying them less tedious, but also to allow state and sequence generation to be performed relatively quickly.

Fortunately the voltage regulators and consumers on the Enzian are relatively well-behaved (as Schult [2]) has previously noted), which allows for these restrictions without impeding our ability to model and control an Enzian.

We start out by requiring that component states have a name that is unique within that component, and that between these states there are "transitions" that describe how, and if, one state can be reached from another. In this thesis we will not investigate a model where states can have arbitrary, or absent transitions between them, instead ruling that the states of a component form an ordered sequence, and that states can transition into

those states that are their neighbours in this order. To transition into a state that is not the direct neighbour of a state, it must successively transition into the neighbouring states between the source and target state.

Each of these "transitions" is made up of a sequence of "transition step sets", each of which is a set of "transition steps".

Two neighbouring states must either have at least one differing input requirement, or at least one of them must have some meta-information attached to it that makes it clear that some action must be taken to transition into it. The second case is for the "Configured" state that we introduce later on, which actually mandates that it has the *same* input requirements as the previous "Powered" state, but is meaningfully different from it in that the component has to be configured, via an explicit action, to reach the "Configured" state, and deconfigured to return to the "Powered" state.

Transition steps always relate to input requirements, and both elements of a transition step must refer to the same input. Component outputs, according to our interpretation, do not change in-between states, only when we fully transition into a new state. A transition step from state a to state b for input i has two components: an optional source requirement and an optional target requirement. The source requirement describes the input requirement that the source a imposes on i, and the target requirement the same for the target state b. The optionality of the source and target requirements makes it possible to have states that "don't care" about an input's value, which is important for encoding the difference between a component that requires that an input has a specific value right before and after it changes, and a component that only imposes the requirement that the input has retained a value until this step (a (Some requirement, No requirement) transition), or has reached a certain value after it (a (No requirement, Some requirement) transition).

A transition step fulfills the invariant that the source requirement differs from the target requirement. Between two states that have the exact same input requirements there thus does not exist a transition step, unless they carry some meta-infomration as previously defined.

We notice that most of the complex states and transitions are lost on the Controllers and Regulators of the Enzian, so we restrict them to a specific subset of states. We call these states "Knuselian", because they were formalized in Knüsel [3].

- "Off", in which a Controller/Regulator is as turned "off" as it can be.
- "Powered", describing the common in-between when a component is not "off", but also not doing anything useful yet.
- "Configured", which is an abstract state into which a component can be "configured", usually through I^2C. The "configured" state may not differ in requirements from the "powered" state and may not assign voltages different from "Powered". We model an internal "configured' PET-wrapped boolean value to differentiate between a configured and unconfigured component. We can assume that a component is not configured if it is not producing any output.

  The "configured" state is also the minimum state that a component must be in for us to consider any monitors it defines to be accessible.
- "On", where they may *assign* voltage to their outputs.

We order these restricted states from least ("Off"), to most ("On") active.

The difference between Controllers and Regulators is principally that controllers may change their outputs at any time, while we require Regulators to re-enter their "Configured" state, be re-configured and may only then turn on again.

We now introduce the third type of component: the *Consumer*. Consumers have no outputs, they are "leafs" of the power tree and only "consume" the power provided to them, as far as we are concerned. Unlike Controllers and Regulators, we allow the specification of arbitrary states for Consumers, and for now arbitrary transitions between those.

Our Controllers, Regulators and Consumers are now all roughly "Knuselian" as in [3].

Compared to the Knüsel [3], we restrict our states in three major ways, one of which was already mentioned previously:

- Every state must always explicitly assign to every one of the components outputs.
- All states must have either one or two "neighbour" states. At most two may only have one. This restriction removes "choice" from the sequencing, which is one source of potentially exponential complexity growth.
- Over all their states, each requirement/output may only change *once* per port. This includes changing from "No requirement/assignment" to "Some arbitrary value".

These may seem like severe restrictions, but they allow for state and sequence generation to be significantly simplified, which we argue is required for online, dynamic power management.

Supporting Change

(Un)fortunately for us, we cannot be content with a static view of the platform. We have to be able to track and expect changes.

The previously introduced Section 7.3.1 values are our proposed solution to at least part of this problem.

From the models perspective, every conductors state value and every components "configured" value has a PET state.

This will allow us to differentiate between most of their various states as they change.

?? shows all the states we expect to encounter while a value transitions. Evidently we need some way to differentiate between the state of "take an action and *Set*" and "take an action and *Read*", which follow right after each other.

For this purpose we attach to each PET state a boolean *NextRead*, which should be true whenever the next action to take is to *Read*.

7.6 Platform State Transition Manager

The full name of this component should be be "Platform State Transition Manager", as it does not manage lower-level transitions directly. For brevity's sake, we will refer to it as "transition manager".

The transition manager is in charge of the successful transition of the platform from one state into another.

The first factor that makes the transition manager's work non-trivial is the fact that the target states the transition manager is supplied with are underspecified, "CPU on" for example, and have to be resolved to a complete target platform state that assigns states to all components and value ranges to all conductors.

The second is that finding a valid sequence beteween two fully specified target platform states is, in general, a hard problem. Luckily, with the restricted Knuselian states (see subsection 7.5.3) we introduced, the problem becomes significantly easier and less computationally expensive to solve, but remains non-trivial.

In addition, once a transition sequence has been found, the transition manager is responsible for returning to the model a valid step to follow, on request, which also includes keeping some internal state up-tod-date with the model so it knows which step can be executed next.

A note on nomenclature: We use the term "transition" when we talk about a change from one state to another, either two platform states or two component states. A "sequence" is a precise set of steps encoded in a suitable format that allows us to execute a "transition", if the steps in the schedule are followed in-order. Alternatively, we sometimes use the name of the precise datastructure we use to implement our sequence, which is a DAG or a graph, instead of "sequence", or simetimes in combination, as in "sequence-graph". A "schedule" is a datastructure that may contain a schedule and additional information for following and re-generating a sequence. In our case, a schedule keeps track of the steps in the sequence that are "ready" or "completed", and thus allows following a sequence step-by-step.

7.6.1 Target Platform State Resolution

To find a sequence we first must find out where we want to go.

In the abstract, target state resolution is a function that, given a platform model M and a target state T, computes value ranges the conductors take if the platform fulfills the target state T:

$$T(m, t) \to C \times (\mathcal{R} \times \mathcal{R})$$

If the mapping

$$State \mapsto Conductor \times (\mathcal{R} \times \mathcal{R})$$

i.e. the restrictions imposed on a conductor by some state, is computationally cheap to compute, and in our case it is, we can take an intermediate step and find the target *states* components must be in, before mapping these onto conductor range restricitons.

Knüsel [3] use something similar to the second approach, but replace a cheap mapping function with a more powerful OMT solver. They split sequencing and target state finding into two steps because using a single OMT instance would take too long, presumably due to some superlinear complexity growth with respect to problem size.

We are motivated similarly: Because we identify a cheap way to find a sequence (see subsection 7.6.2) from a present, known, platform state to some valid target platform state we also use two steps to find a sequence.

Target Platform State Precomputation

We describe the first step in our target platform state generation algorithm, which precomputes target states offline, one for each component state.

See Section 7.6.1 for the second, online fusion step.

The idea behind this approach is to minimize the time it takes to generate a target platform state once we have precomputed the target state cache. Overall, we can expect most platforms to be stable almost all of the time, so expending the precomputation effort initially makes sense.

Our approach works like so:

For each state s_i we can precompute the platform state. We sidestep the complexity of the nested and potentially recursive manual resolution by modelling platform state resolution as a MIS problem.

Intuitively, we create a graph G with a vertex per state $s \in S$ and connect incompatible states, as well as all states of a single component. We then find a MIS on G. If the MIS has size $|C|$ then the vertices in the MIS assign a single state to each component without violating any incompatibility restrictions.

Note that since G contains $|C|$ connected components a *maximal* independent set of size $|C|$ is also a *maximum* independent set of G.

We augment our intuitive description with a more formal one:

Let $G = (V, E)$ be a graph with $V = \{v_{c,s} \mid c \in C, s \in c.states\}$.

We add edges connecting the states of each component:

$$E_S = \{(v_{c_1,s_1}, v_{c_2,s_2}) \mid c_1 = c_2 \wedge s_1 \neq s_2\}$$

And connect states that are "incompatible" according to some incompatibility predicate P_I

$$E = E_S \cup \{(v_{c_1,s_1}, v_{c_2,s_2}) \mid P_I(c_1, s_1, c_2, s_2) = true\}$$

We can now re-use G to solve arbitrary component target states $T \subseteq \{(c, s) \mid c \in C \wedge s \in c.states\}$ like so:

We create a new graph $G_m = (V_m, E_m), V_m = V, E_m = E$ where all component target states $v_{c,s}, (c, s) \in S$ are pre-marked, i.e. added to the future-MIS set M already.

We complete the MIS M on G_m. If $|M| = |C|$ then we extract a state assignment

$$target[c] = s : v_{c_s,s} \in M \wedge c = c_s$$

To increase confidence in our solution we prove two two theorems about our solution, showing that if we find a solution, it is correct:

Theorem 2. $|M| = |C| \implies$ *M only assigns a single state per component.*

Proof. Trivially, because all $v_{c,s}$ states of a component c are connected no MIS M can assign multiple states to the same component, as otherwise the set is not independent. □

Theorem 3. $|M| = |C| \implies P_I(c_1, s_1, c_2, s_2) = false$ *for all* $v_{c_1,s_1}, v_{c_2,s_2} \in M$.

Proof. Trivially, if $P_I(c_1, s_1, c_2, s_2) = true$ for some $v_{c_1,s_1}, v_{c_2,s_2} \in M$ then there exists an edge between v_{c_1,s_1}, v_{c_2,s_2} by construction and no independent set on G can exist containing both V_{c_1,s_2} and v_{c_2,s_2}, which was our assumption. □

We must unfortunately leave the proof of the theorem that the existence of a target platform state implies the existence of an MIS, and that we find it, as conejecture.

Incompatiblity Predicate We can now continue by defining our incompatility predicate, that defines which states cannot be enabled at the same time.

As a basline, trivially incompatible are the states of a single component, as a component cannot be in multiple states at a time. From this we can now iteratively develop the incompatiblity predicate P_I:

$$P_I(c_1, s_1, c_2, s_2) \in \{true, false\}$$

by going through arguments for which it evaluates to $false$.

The predicate takes as its argument two pairs (c_i, s_i) denoting that component c_i is in state s_i. We will usually use $i \in \{1, 2\}$.

Let $c_1 \neq c_2$ and w.l.o.g c_1 has an input i, c_2 has an output o such that $(i, o) \in conductors$. Trivially s_1, s_2 are incompatible if s_2 assigns to o a value that is outside the safe range of i. We also decide that s_1, s_2 are incompatible if s_2 assigns to o a value that is oustide the safe range of o, which effectively makes it impossible to enter state s_2.

For the remainder we assume additionally that s_1 requires that i has some value $r \in [r_l, r_u]$.

s_1, s_2 are trivially incompatible if s_2 assigns to o some range $a \cap r = \emptyset$. And this is the first time we encounter a non-trivial incompatiblity: we have to consider "siblings". Assume sibling component c_s with input i_s such that $(i_s, o) \in conductors$ with some state s_s that imposes a requirement r_s on i_s.

If $r_s \cap r_1 = \emptyset$ then the two states trivially are incompatible. This is a necessary, but sufficient, check because if

$$a \cap \left(\bigcap_{r_i \in R} r_i \right) = \emptyset$$

meaning the assignment a is incompatible with the set of requirements R, there either exists an r_i such that $r_i \cap a = \emptyset$ or a pair r_i, r_j with $r_i \cap r_j = \emptyset$.

The second non-trivial incompatibility involves the unusual $o = InputMatch(i)$ assignment, which states that the output o will, in this state, have the same value as the input i is receiving. We note that $InputMatch$es can be chained. By essentially "passing through" an output from lower down the powernet this introduces exciting new incompatiblity opportunities:

See Figure 7.9 for an overview of the different state incompatibilities of *InputMatch*es.

Whenever we encounter an $InputMatch(i)$ with an output o connected to i, we have to consider all states that assign a value to o, so for each output connected to an $InputMatch$ed

input we can gain new siblings, to which we must apply the same incompatibility check as above.

For *InputMatch assignments* we naturally do two trivial checks: One for the outputs safe values and one for the inputs.

For a terminal assignment a_t to output o_t by some state s_t that does not continue the *InputMatch* chain we can again test for compatibility with the siblings introduced by o_t, but also finally with a_t.

Realization We realize Target Platform State Precomputation as an Integer Linear Program. We could, of course, also implement the target state computation as a SAT problem, which our testing indicates could even be even faster, but using SAT will only give us an arbitrary target state when we are actually looking for a platform state where every component is as "off" as possible. There is, essentially, a hidden optimization requirement baked into the problem. To solve exactly these kinds of "optimization" problems with SAT solvers there has been work on SAT solvers with preferences, but these have unfortunately not been widely implemented and we did not expend the effort of implementing one ourselves. ILP was deemed to strike the best balance between implementation effort and speed, and is thus recommended.

The incompatibility predicate can either be implemented by considering each pair of states and then recursing on *InputMatch*es, or considering each pair of states and then trying to *find* an *InputMatch*-path between the two. We find the recursive approach more intuitive and also likely faster than the purely-pairwise approach.

Online State Fusion

Once all the pre-computation effort has been expended we can now reap the rewards.

Given targets T_i, T_j with precomputed platform states P_i, P_j we can compute a combined platform state $P_\cup$ like so:

Algorithm 7: Target State Fusion

$P_\cup \leftarrow \emptyset$;
for *each component* C **do**
  $P_\cup[C] \leftarrow \max_{i,j} P_i[C]$
end
$C_T \leftarrow$ conductor target ranges;
`/* Sanity check:` `*/`
for *targets* t **do**
  $Ensure(t \in P_\cup)$
end
for *neighbouring pairs of components* (C_1, C_2) **do**
  $Ensure(\{P_\cup[C_1], P_\cup[C_2]\} \notin E)$;
end
for *each component* C **do**
  Ensure that requirements in state $P_\cup[C]$match conductor target ranges;
  Ensure that assignments in state $P_\cup[C]$ match conductor target ranges;
end

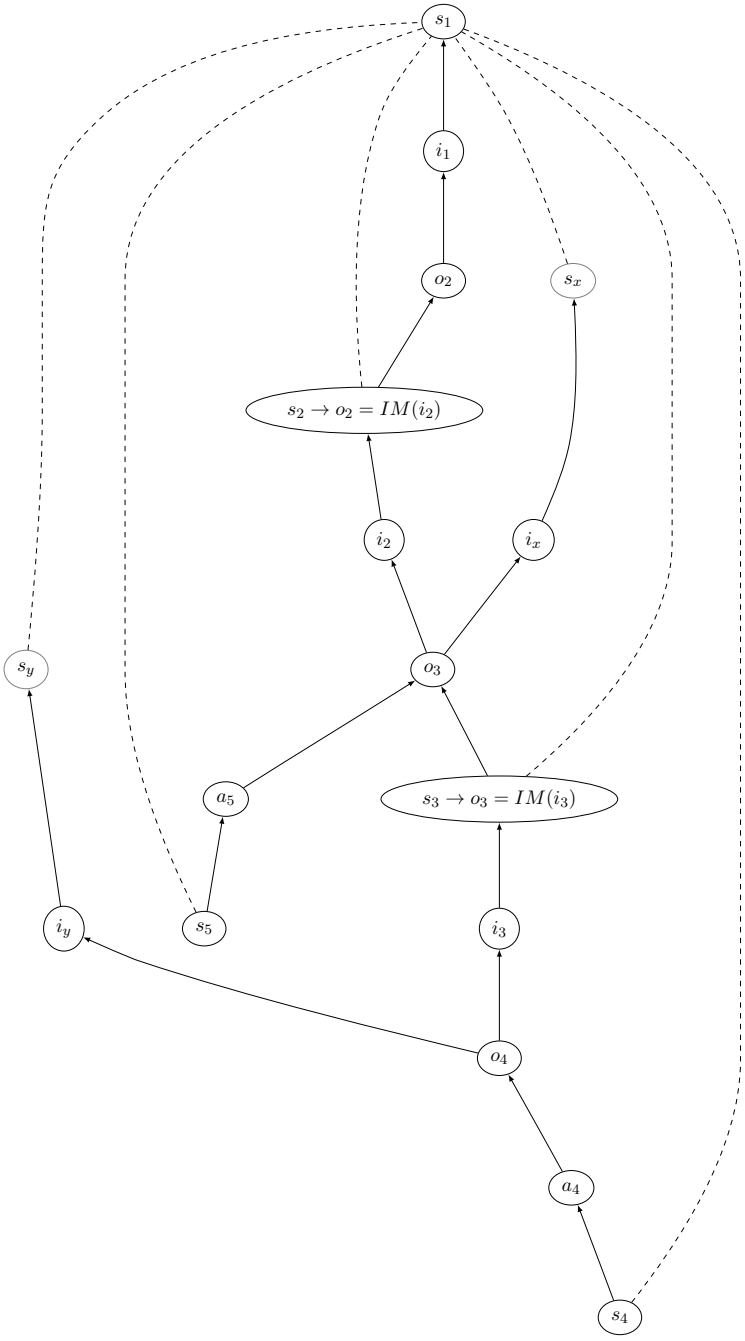


Figure 7.9: Figure showing the possible state incomaptibilities introduced by successive *Inputmatch*es. s_1 is the base-state under consideration. States that can be incompatible are connected with dashed lines. Siblings of s_1 are in gray circles.

In the following we will attempt to give an intuition as to why this approach works and why the "sanity check" is necessary.

Proof. Consider a power tree P with some components $C_i \in P$. We associate with every C_i a range of possible states it could be in given the rest of P, which we call $R_i = [r_{i,1}, r_{i,2}]$.

For some set of restrictions that currently apply to C_i, call them $R_x \in X_i$, we can calculate

R_i as follows:

$$R_i = \left[\max_{R_x \in X_i} r_{x,1}, \min_{R_x \in X_i} r_{x,2} \right]$$

If we artificially restrict the state of some components, say by giving them target states, we are adding new restrictions to X_i and further restricting the state C_i may be in.

Consider now some target platform states T_k. Each of them has associated with it a set of restrictions $X_{k,i}$ for each component i.

If we combine the target platforms states T_k into $T_C = \bigcup_k T_k$ it follows that $X_{C,i} = \bigcup_k X_{k,i}$ and

$$R_{C,i} = \left[\max_{R_x \in X_{C,i}} r_{x,1}, \min_{R_x, \in X_{C,1}} r_{x,2} \right]$$

But crucially,

$$\begin{aligned} R_{C,i} &= \left[\max_{X_{k,i} \in X_{C,i}} \left(\max_{R_x \in X_{k,i}} r_{x,1} \right), \max_{X_{k,i} \in X_{C,i}} \left(\min_{R_x, \in X_{k,i}} r_{x,2} \right) \right] \\ &= \left[\max_{T_k \in T_C} \left(r_{k,i,1} \right), \max_{T_k \in T_C} \left(r_{k,i,2} \right) \right] \end{aligned}$$

And finally

$$F_i = \max_{T_k \in T_C} \left(r_{k,i,1} \right)$$

is exactly how we compute our fused state F_i, and so as long as

$$\max_{T_k \in T_C} \left(r_{k,i,1} \right) \leq \max_{T_k \in T_C} \left(r_{k,i,2} \right)$$

i.e. there exists a valid target platform state at all, our state fusion return a valid platform state.

This confirms that our approach *can* generate a valid platform state, but also explains why we have to check if the platform state we generated is actually valid.

□

7.6.2 Sequences

Single-transition Sequences

Similar to target state computation generating single-transition becomes, comparatively, simple under our restrictions.

For sequences where every conductor only changes – or "transitions" – once, we can iteratively build a schedule-DAG $S = (V, E)$ that orders conductor and component state changes.

We begin by adding to our schedule graph two vertices per changing conductor $c_i \in CC$: one indicating that the conductor is permitted to change, $c_{i,p}$, and one indicating that the conductor has already changed, $c_{i,h}$.

$$V = V \cup \{c_{i,p}, c_{i,h} \mid c_i \in CC\}$$

A conductor has to change strictly after it is permitted to do so, so we can add our first sequence requirement:

$$E = E \cup \{(c_{i,p}, c_{i,h}) \mid c_i \in CC\}$$

And already we only have two more things to consider: restrictions imposed by state transition requirements and restrictions imposed by state assignments.

For every component d_j with $s_p = state_{present}(d) \neq state_{target}(d) = s_t$ we add two vertices per state s_i: one indicating that the state s has been reached, $s_{j,i}$ and a second one we'll use for sequencing that requires state s_i to be reached, but is not required for the component to reach s_i itself, $s_{j,i,post}$. Of course *post* happens strictly after the proper state has been reached.

$$V = V \cup \{s_{j,i}, s_{j,i,post} \mid d_j \in D, s_i \in states(d_j)\}$$
$$E = E \cup \{(s_{j,i}, s_{j,i,post}) \mid d_j \in D, s_i \in states(d_j)\}$$

We first deal with the restrictions imposed by assignments. For a component d_j with $s_p = state_{present}(d_j) \neq state_{target}(d_j) = s_t$ by convention every state $s : s_p \leq s \leq s_t$ assigns a value to every output of d and the conductor c_i connected to that output. If $c \in CC$, then there are states $s_m, s_n : s_p \leq s_m < s_{m+1} \leq s_t$ that assign different values to c_i.

For our first proper sequence requirements we can now say that the conductor change of c_i has to be permitted strictly before any transition steps from s_m to s_{m+1} are executed, because we do not know which one of these will actually cause c_i to change, only that it has changed once s_{m+1} has been reached.

$$E = E \cup \{(c_{i,p}, s_{j,m,post})\}$$

And conductor c_i only changes once s_{m+1}, which actually assigns the target value to c_i, is active:

$$E = E \cup \{(s_{j,m+1}, c_{i,h})\}$$

We know that s_{m+1} is the state that assigns the targeted value to c_i, because the assignment the to a conductor is only allowed to change once between two states of a component.

A special case that is only necessary for some interpretations of the schedule graph are controllers that do not change state. Regulators in the same situation are unable to change their output, but controllers can. If the schedule graph is expected to contain all assignments as edges as well as vertices, which is not required, or to be made more readable for debugging, then for each controller that does not change state we add a vertex for that state and connect it to the $c_{x,h}$ conductors that the controller state assigns to.

Having dealt with the assignment of conductors we can finally move on to discussing the state transition requirements. A component can of course only be in a single state at once, and a strict order on component states thus has to be enforced.

We call the changes necessary to go from one state to the next "transitions" as a whole, and the atomic changes happening to the requirements of a single input "transition steps" or

"steps". For example, for two states s_1, s_2, with s_1 requiring input $i = 0V$ and s_2 requiring $i = 5V$ we can encode this as a transition step from s_1 to s_2 with $(i, 0V, 5v)$. Note that it is possible for a step to involve at most one "None" requirement like $(i, 5V, None)$ or $(i, None, 5V)$ to accomodate situations where

For regulators and controllers we rule that all transition steps between two states can happen in any order and still be a valid. For consumers, however, this does not hold. The ThunderX CPU on the Enzian, for example, imposes transition restrictions in "groups", which we will now call "step-sets". The steps within a set may happen in any order, like an unordered set, but the step-sets within a transition must obey the order they are defined in. For controllers and regulators we model their steps as being part of a single step-set.

To illustrate how we can enforce these sequence requirements in our DAG we consider an example transition from s_1 to s_2 with two step-sets $P_1 < P_2, P_1 = \{p_{1,1}, p_{1,2}\}, P_2 = \{p_{2,1}\}, p_{1,1} = (i, None, Some), p_{1,2} = (i, Some, Some), p_{2,1} = (i, Some, None)$.

For the sake of brevity we use $a < b$ as $E = E \cup (a, b)$ and omit the explicit adding of vertices to V.

We first add four vertices $s_{1,post} < start_{P_1} < end_{P_1} < start_{P_2} < end_{P_2} < s_2$ to enforce inter-stepset order and that they happen in-between the two states.

For steps with no initial requirement like $p_{1,1}$ we simply enforce that they happen *before* their stepset ends, because the "None" requirement indicates that we do not care about the value of the input *before* this stepset, only after: $p_{1,1} < end_{P_1}$. For steps with no terminal requirement like $p_{2,1}$ we apply the same principle in reverse: $start_{P_2} < p_{2,1}$, as this step does not care about the value of i *after* its execution.

If the step is a command-step then we add a vertex denoting it as such and use use that in-lieu of the step above. If the step concerns a conductor changing, then for $p_x < end_{P_y}$ we use c_h, i.e. the conductor has to change before the end of stepset P_y, and for $start_{P_x} < p_y$ c_p, i.e. the conductor is only permitted to change after the start of the stepset P_x.

Note that $None, Some$ will occur primarily while turning a system on, as some components will not care about the value of conductors when they're off, and the reverse when turning the system off. In both cases, the target platform state still tells us the exact value the conductor will, or should, have deduced from the assignment to the conductor in the final state, which cannot be "None".

If two consecutive states do not have any stepsets between them we can simply connect them directly as $s_{1,post} < s_2$.

If at any point during this procedure the schedule graph should cease being a DAG, which can happen for some target states or due to misconfiguration, then scheduling has to be considered failed.

Compared to prior work, which emits static schedules with a fixed order, producing a schedule DAG has the benefit of enabling, in theory, parallel execution of the schdule in dynamic orders. This has the potential to speed up the execution times of the schedule if there are components that take a long time to react. As Knüsel [3] notes, this is only really true for the bring-up of clock generators on the Enzian, with most other regulators reacting very quickly, and as we are unconcerned with the overall execution time of the schedule at this tage anyway, we do not investigate the effects of this further, instead keeping the schedule in DAG-form primariliy for flexibility.

Stepset Example To further illustrate the need for stepsets with a practical example, that is also useful as an example of how unpredictable and difficult to model hardware can be.

We begin by quoting Schult:

> [...] It is also not clear how forgiving the bootstrap process of the ThunderX is to the main clock being enabled a bit later than VDD IO is powered. For this reason, we might be tempted to decompose this step as follows:
>
> 1. Enable the main clock
> 2. Power VDD_IO
>
> The design of the Enzian platform does not allow this, however: The main clock generator is powered by the conductor connected to the VDD IO input of the ThunderX. (Schult [2])

This also means that the clock generators can only be allowed to enter their "Powered" state if we also transition the ThunderX into its "Reset" state, and so a valid platform state where we *only* power the clock generators does not exist.

What can also be derived from the quote above, is that

$$\text{Enable the main clock} \leq \text{Power VDD\_IO}$$

but

$$\text{Enable the main clock} \not< \text{Power VDD\_IO}$$

and so we arrive at

$$\text{Enable the main clock} = \text{Power VDD\_IO}$$

In effect, this means that because the main clock is only enabled *when* VDD_IO is powered that we must, in the strictest sense, violate the sequencing requirements of the ThunderX, due to the way the Enzian is configured.

In practice, the clock generators seem to be able to start generating output quickly enough after bein provided with power that the ThunderX can still consider the two events to happen "at the same time".

We can encode this the same way Knüsel [3] do, by having a step set

$$[Enable(VDD\_IO33), Enable(PLL\_REF\_CLK)]$$

and implicitly allowing VDD_IO33 to happen *after* PLL_REF_CLK, if required.

Sequencing Failure Unfortunately, the schedule generation cannot escape the responsibility of continuously checking if the DAG contains a cycle, or doing so at the end of the process, because sequencing can fail for *valid* target combinations, as we intend show here.

There is no guarantee that all *valid* target combinations are also reachable through a sequence.

See Figure 7.10 for an illustration of the following example.

Take consumer T with two states T_1, T_2 and two steps from $T_1 \to T_2$: T_{s1}, T_{s2}, add two regulators R_1, R_2 and two boolean conductors C_1, C_2.

R_1 has an output connected to C_1 and R_2 to C_2.

The two regulators have two states each, on, off. In the on state R_1 assigns to C_1 and R_2 to C_2 $true$, in off $false$.

The target platform state with C_1, C_2, T all being on is trivially a valid state.

We require that the two steps of the consumer T happen in-order:

$$T_{s1} < T_{s2}$$

Step T_{s1} needs $C_2 = true$, T_{s2} needs $C_1 = true$

$$C_{2,h} < T_{s1}$$
$$C_{1,h} < T_{s2}$$

And lastly to step from off to on R_2 requires that $C_1 = true$.

$$C_{1,h} < R_{2s}$$

We now see the cycle:

$$R_{2s-end} \rightarrow R_{2-on} \rightarrow C_{2,h} \rightarrow T_{s1-end} \rightarrow T_{s2-begin} \rightarrow C_{1,h} \rightarrow R_{2s-end}$$

and have shown a sequencing failure from a valid target platform state.

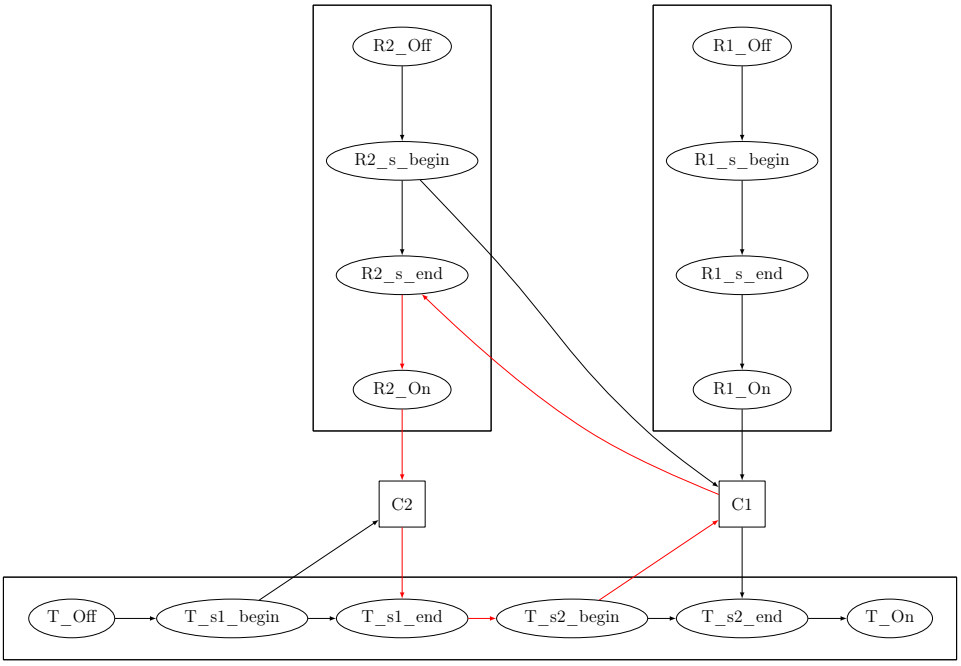


Figure 7.10: Sequence of a platform with a valid target state in T_On but no sequence leading to it. The cycle is marked in red.

Composite Sequences

Unfortunately the assumption that there exists a sequence such that every conductor only changes once does not hold for all target states, trivially when a regulator has to change the value it outputs and has to be reconfigured. In this case we have to composit multiple sequences into a larger one, with each individual sequence only changing each conductor once.

We conjecture that, for the restrictions we impose on our platform, a conductor requiring multiple changes always involves a regulator having to change its output.

Determining if compositing is required under these restrictions is thus very simple: we can iterate over all regulators that are currently assigning to a changing conductor and check if their new target assignment differs from their previous one and if that would require reconfiguration. If yes, then we must composit.

Once all regulators that must change their output are collected we can compute for each of them the state that we require them to return to in order to be reconfigurable.

As we already use an ILP solver for target platform state computation we can use a very similar approach here: we find a target platform state with the same restrictions as usual, except that for our targets we don't require that their components are in those *exact* states but instead in *at most* those states.

For parameters we use the reconfigurable states we described above, and generate an intermediate target platform state.

Because this intermediate target platform state is necessarily "lesser" than our current platform state we can sequence to it without requiring further composition. Once we have reached the target platform state we generate a sequence to our original target platform state.

Sequence Invalidation

While we are following a sequence it is possible that it becomes invalidated by some change in the platform state. This should only occur rarely when a component

An invalid sequence can be repaired, even if the target state changes, but we argue that this is not worth the additional complexity, and so we invalidate sequences when these conditions are met.

Sequence Repair Instead of invalidating a sequence and discarding it altogether it is possible that sequence repair, a term borrowed from planning where plan repair is sometimes used to recover information from plans that no longer fit the objective exactly, can be achieved for our circumstances and for complex instances even be worth it.

We provide a rough outline of a potential algorithm to repair sequences, to demonstrate roughly what we conjecture should be possible, but cannot provide further insight into the

problem.

Algorithm 8: Sequence Repair

```
if target platform state changes then
    Recompute target state;
    for all components connected to changing conductor do
        Recompute steps;
    end
else
    for all components that changed state do
        Recompute steps;
    end
end
```

SCRAM

A SCRAM is a known sequence to return the platform, not matter its current state, into a known safe state "forcefully". The most aggressive variant is to cut power at the root of the tree, which on the Enzian is the PSU. A SCRAM could also be implemented as a simple forced target platform state that we then let our normal sequencing infrastructure deal with. This approach is problematic primarliy because this sequencing, especially in circumstances where a SCRAM is an appropriate reaction, may not be able to find a sequence to the SCRAM target. Ideally, a SCRAM sequence is pre-generated, iterating through the voltage regulators and commanding them to cease outputting power immediately in a known, safe order, ignoring all alerts and faults and continuing unabated until the platform has reached the desired known, safe state.

For our purposes we favour a pre-generated sequence that

1. Set all fans to max speed
2. Turn the platform off iteratively

which is implemented in the current Enzian power manager as a "fan scram".

7.6.3 Operation

The transition manager keeps state in the form of

- Optionally, a schedule
- The progress on that schedule, in the form of sequence steps that are "ready", or "completed".
- Optionally, A target state cache
- Model requested (underspecified) target states
- User-requested (underspecified) target states
- Optionally, the current target state.

The transition manager has a relatively simple interface, taking as input:

- New target states

- Sequence invalidation commands
- Cache invalidation commands

And supports two interfaces: one for returning a *step* for the model to follow, another for synchornizing the schedule state with the model.

Target states are kept in a stack, and new target states are added to the top of the stack. The reason for using a stack is that components will want to enter temporary "intermediate" states, but should not have the authority to override the underlying wishes of other components.

When the transition manager is asked to return a step but either has no sequence then one of the target state stacks is popped and the result made the current target state. The transition manager prefers the model-requested target states, because these are either in service of another user target, or are ensuring the proper functioning of the platform, for example in reaction to a fault.

Before generating a sequence, the transition manager checks if the target states have already been reached. If all of the target states in the set have been reached, the target state is dropped and the next one selected.

The following pseudo-code (algorithm 9) illustrates how the transition manager returns a step for the model to follow.

Algorithm 9: Transition Manager: get_step()

```
if schedule = None then
    if state_cache = None then
        regenerate_target_state_cache();
    end
    if current_target = None then
        if ¬ model_request_stack.is_empty() then
            current_target ← model_request_stack.pop();
        else if ¬ user_request_stack.is_empty() then
            current_target ← user_request_stack.pop();
        else
            current_target ← None;
        end
    end
    if current_target = None then
        return None;
    end
    schedule ← generate_sequence_graph();
    for step s ∈ schedule do
        if schedule.sequence_graph_step_is_ready(s) then
            mark_ready(s);
            schedule.ready_steps.push(s);
        end
    end
end
return get_step_from_sequence_graph();
```

get_step_from_sequence_graph() must be a deterministic, idempotent function, meaning it does not alter the state of the transition manager and always returns the same step, given the same sequence graph.

The transition manager as envisioned does not include a timeout or cap on the number of times a step is retried, though a basic version of either of these features would be trivial to implement. More advanced solutions would run into the issue that a proper timeout or maxium number of retries is both a matter of the sequence/schedule being followed and the component-DES implementation. One option would be to simply accept this distributed responsibility for retry limits, but we did not investigate this in sufficient depth to offer a conclusive opinion on whether this solution is good enough or not.

The model controller is responsible for asking the transition manager to synchronize itself with the current model state. When it does, the transition manager follows these steps to get up-to-date with the model state:

If the transition manager is currently following a schedule, it checks all steps that are currently ready for execution. For each of these ready steps, the transition manager checks with the platform model if the step post-conditions have been completed, for example if a conducor's value is in a certain range, a component in a certain state, or configured. All the ready steps that have been completed are marked as "complete" instead of "ready". This process is repeated until the transition manager finds no more ready steps to complete. Completed schedules are deleted. See algorithm 10 for pseudo-code implementation.

Algorithm 10: Transition Manager: synchronize(model)

```
if schedule ≠ None then
    completed_steps ← {};
    do
        completed_steps ← {};
        for s ∈ schedule.ready_steps do
            if model.check_post_condition(s) then
                mark_completed(s);
                completed_steps.push(s);
                schedule.ready_steps.remove(s);
            end
        end
        for s ∈completed_steps do
            for n ∈ sequence_graph.children(s) do
                if schedule.sequence_graph_step_is_ready(n) then
                    mark_ready(n);
                    schedule.ready_steps.push(n);
                end
            end
        end
    while ¬ completed_steps.is_empty();
    if schedule.ready_steps.is_empty() then
        transition_manager.delete_schedule();
    end
end
```

Note that the transition manager cannot ensure that the steps have *been executed* in the correct order, it can only release steps that it knows *should be* execute next.

7.7 State Transitions

Given our model of the platform we can now "induce" change on three levels:

1. State changes to a single PET-wrapped value
2. Sequences with at most one change per conductor
3. Composite Sequences, made up of Single-change sequences

We will see that we can realize the first type, changing the state of a single PET-wrapper value, "atomically" in some sense and that we can build sequences out of these atomic changes such that we never leave our model in an inconsistent state, barring faults or unpredictable behaviour.

To generate these more complex sequences we have to resolve the, potentially underspecified, user-provided target states to a full platform target state (see subsection 7.6.1) and then generate a sequence from the current state to the target platform state (see subsection 7.6.2) in a secondary step.

7.7.1 PET State changes

We can induce state change by exploiting the behaviour of our model: we simply set the target of the PET value we want to alter and ensure that $NextRead = false$.

Chapter 8

Evaluation

We evaluate the part of our solution we implemented, which is the target platform state and sequence generation.

8.1 Scaling and Online-Feasibility of Sequencing and State Generation

Because online, dynamic management is more time-sensitive than its offline counterpart, we want to show that our target platform state and sequence generation can operate withing stricter time-limits than prior work could, and ideally quickly enough for online management.

Even given our general assumption that in-between faults the platform is well-behaved a solution that can react to platform changes or user requests with a lower latency is desirable both from a user-experience standpoint, as well as for the marginal improvements in platform safety that reacting in 0.5 instead of 5 seconds can provide in some circumstances where a future fault is avoided.

8.1.1 Setup

We provide a comparison to the optimizing off-line implementation from Knüsel [3], which heavily influenced our own way of modelling the platform, and consider it our baseline. Even though [3] use an optimizing solver to find "optimal" platform states our own implementation, while less flexible, implements similar optimizations. For example, we also require that states are "minimal" and compute arguably optimal conductor limits.

For time reasons, it was unfortunately not possible to also compare against Schult [2]'s approach.

We appropriate a test case from Knüsel [3] for comparison, where they generate platforms with n duplicated ThunderX-CPUs, like the CPU used on an Enzian, and their power network, like on an Enzian plus some shared infrastructure. We force this platform to be all-off and then ask it to turn on (the CPUs are "On") and the reverse; turning it off (the CPUs are "Off") again.

It is important to note that our implementation is written in Rust and partially primed for speed, while [3] use Guile-Scheme, admittedly interact with the Z3 SMT solver inefficiently,

and were not written with online-management in mind. Nevertheless, we do not believe that improvements on these details could provide the order-of-magnitude speedups we will be able to demonstrate.

In addition to the restrictions inherent in our solution, the implementation assumes in addition:

- all conductors change, i.e. we have an easy way of telling which conductors do change when going from the present state to the target state.
- the platform is trivially observable, i.e. a conductor carrying voltage implies that we can monitor it. Our baseline does ensure this,

Knüsel [3]'s implementation includes some advanced functionality that we do not support, branching state transitions, for example.

For output, [3]'s implementation generates a file with ordered *configure, set-to and monitor* commands, while ours outputs a more abstract sequence graph and accompanying target conductor ranges, from which the same information can be extracted. We argue that the effort to translate our output to a format similar to the one from [3]

We run this evaluation on a Dell workstation with a 4-core x86 Intel Xeon E3-1225 and 32GB of DDR4 RAM for scaling and baseline comparisons with [3] and additionally on an actual Enzian BMC, a 2-core Zynq 7000 ARM processor with 1GB of DRAM to show actual production feasibility.

Aside from the default "Full-Cache" variant of our implementation, which precomputes the entire target state cache, we add a second variant "On-Demand-Cache", which still computes target platform states for individual targets and adds them to the cache before fusing them, and finally a "No-Cache" variant, which does not use the cache at all, and solves the target platform state problem for all targets at once.

For our solution and the Precision Desktop, for every variant we run 5 iterations per platform size, from 1 CPU up to 100 CPUs or when a single iteration takes longer than 120 seconds to compute, whichever comes first. We generated "Full-Cache" up to 15 CPUs, "On-Demand-Cache" up to 73 CPUs, and "No-Cache" up to 100 CPUs. For our solution and the Schibenstoll, for every variant we run 5 iterations per platform size, from 1 CPU up to 25 CPUs or when a single iteration takes longer than 120 seconds to compute, whichever comes first. We generated "Full-Cache" up to 8 CPUs, "On-Demand-Cache" up to 25 CPUs, and "No-Cache" up to 25 CPUs.

We generate for most of the baseline ("all-on" state, "on" and "off" sequences) results up to 32 CPUs, and compute the "initial" state for up to 100 CPUs.

8.1.2 Results & Interpretation

To explain our findings we will take a tour through the stages of generating a sequence that takes us to an underspecified target platform state. We begin with the precomputation step, continue into the step where we actually generate the target platform state and finish with sequence generation. Along the way we also compare combinations of these steps to our baseline.

See Figure 8.1 for a comparison between "turn-on" and "turn-off" times, showing that "turn-off" is relatively more difficult for our solution, but less so for the baseline. For our implementation, we cannot make out a consistent difference in difficulty between generating

an “Off” vs an “On” target platform state, as per Figure 8.2. We fill focus on the “turn-on” state generation/sequencing performance of our solution, which appears to be more difficult overall to sequence for our solution than “turn-off’.

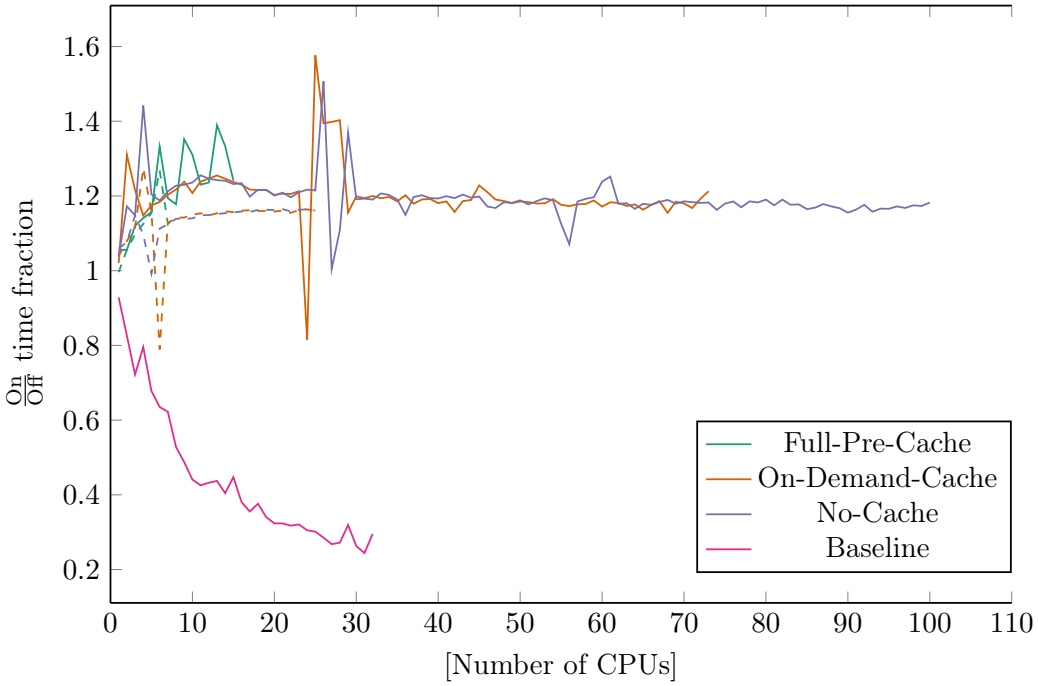


Figure 8.1: $\frac{On}{Off}$ sequencing time ratio. Dashed lines show data from Schibenstoll01, the undashed lines are from the Precision desktop, expect for the pink line, which shows baseline data. The ratio between sequencing “On” and “Off” seems to increase, but then settle at about 1.15, consistently for sequencing on the Schibenstoll01, as well as on the Precision desktop. Interestingly, for our baseline this reverses, and sequencing “Off” takes much longer, up to an observed 3 times, than sequencing “On”. We can conclude that there is no inherent difference in the “Difficulty” of sequencing On or Off sequences, and that instead different implementations can find one or the other much easier to solve for. We also interpret our data to mean that this difficulty ratio converges to a constant, at least for our solution, for sufficiently large problem sizes and that a runaway effect is unlikely.

The first stage in our solution is “Precomputation”, where the MIS-problem object is initialized with shared constraints (single-state components, incompatible state combinations) and the Full-Cache variant additionally caches the platform state for every feasible component-state combination.

As expected, the Full-Cache precomputation takes orders of magnitude longer than the other two variants, which take roughly the same amount of time, see Figure 8.3a.

However, when we look at time taken for generating the completed results — which for Full-Cache just means state fusion, for the On-Demand-Cache state generation and fusion, and for the No-Cache variant simply generating the target state — Full-Cache outperforms the other variants by factors of roughly 2-10 for smaller platform sizes between 1-10 CPUs, see Figure 8.3b and Figure 8.3c. Predictably, for platforms with only a single CPU the Full-Cache variant does not have to do any fusion at all, and simply returns the platform target state from the cache.

For 10 CPUs, Full-Cache takes roughly 30 seconds, which we feel would still be acceptable

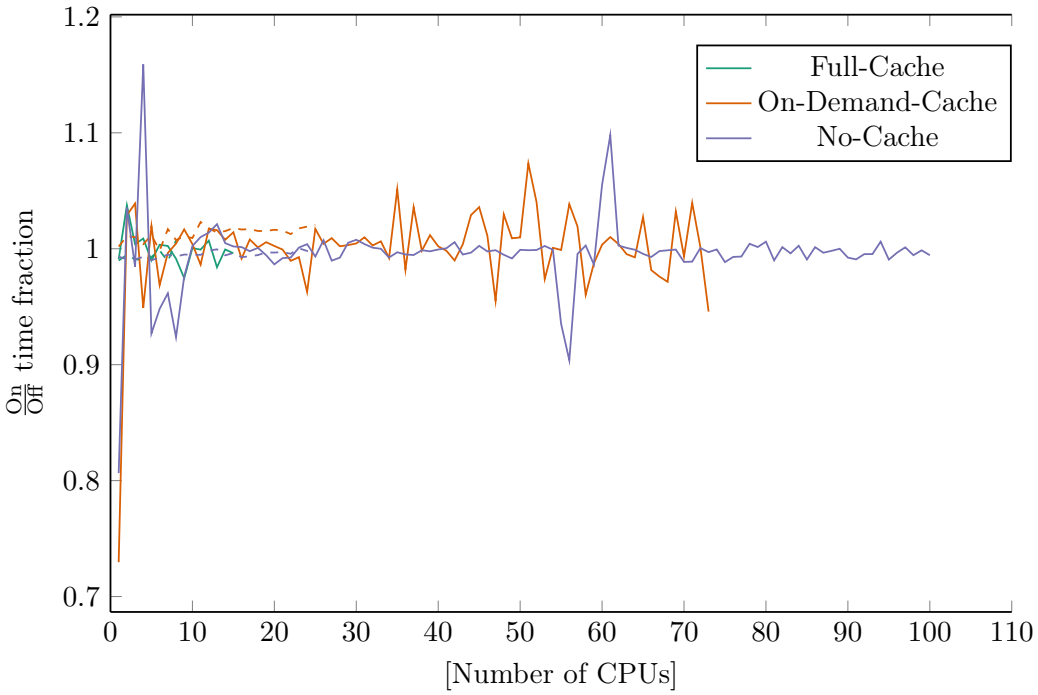


Figure 8.2: $\frac{On}{Off}$ state generation time ratio. Dashed lines show data from Schibenstoll01, the undashed lines are from the Precision desktop dataset. We observe that there is almost no consistent difference between generating an "On" vs an "Off" state, barring values we can confidently consider noisy.

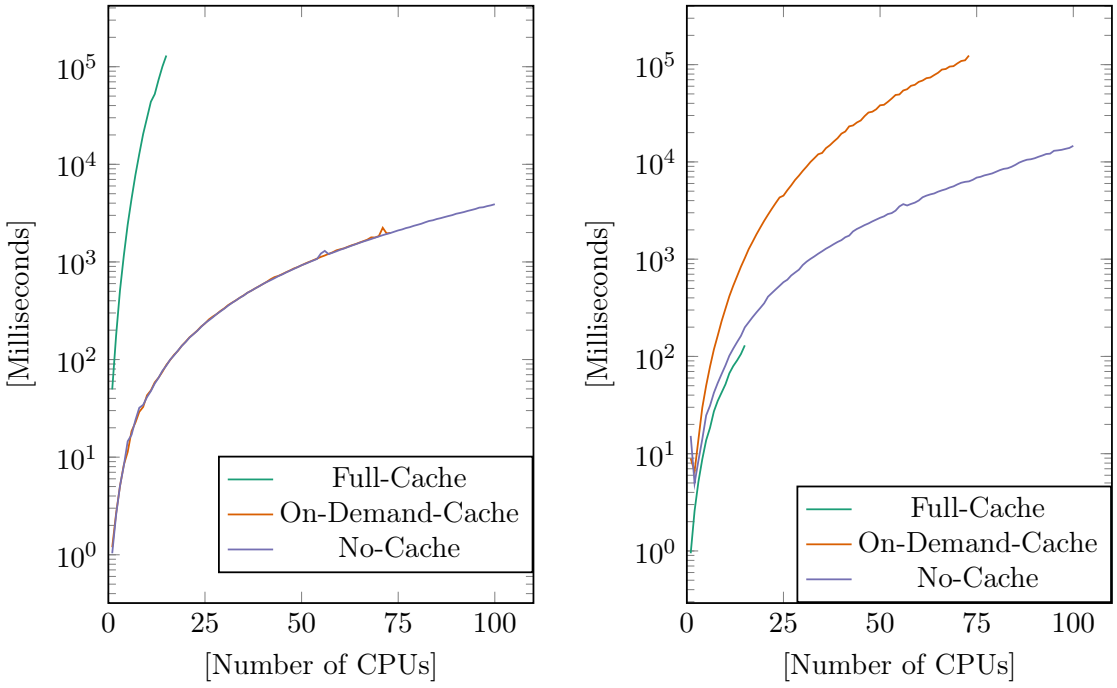
as a one-off cost for a stable platform that requires very low latencies. Ideally, it seems, the cache is initially generated on-demand, but also continuously generated in the background while the system is idling.

Combining the precomputation and finding platform target steps we end up with Figure 8.4. This is sufficiently close to the target generation steps for our baseline, which has to generate both an initial and all-on/all-off state. Because the baseline requires both these states to generate a sequence, while ours would rely on information otherwise collected to calculate the current platform state, we combine their times for a single "combined baseline state generation" metric.

Because the "all-on" target makes up the bulk of this combined vaule, compare Figure 8.5, we also show the "initial" target generation. "Initial" is a state where all components are "Off" and is especially easy for the SMT solver to solve for, because the target state overlaps with the target metric: wanting the components to be as "Off" as possible. We include it as a best-case example for the baseline state generation.

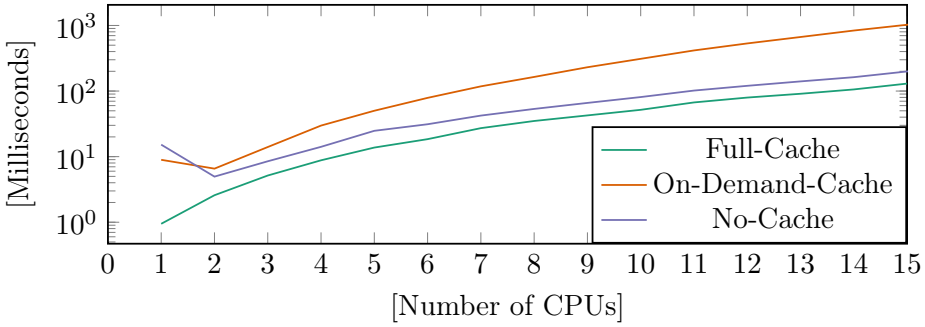
Even so, only for large platforms with 6 or more CPUs does the baseline outperform the Full-Cache variant, which has already done most of the work for all possible target states. The other two variants outperform the baseline by 1-2 orders of magnitude throughout, thanks to our additional restrictions/assumptions and avoidance of complex SMT-/OMT-solvers.

If no caching at all is wanted, or the platform size grows very large, then explicitly solving the target platform state for all targets at once, No-Cache, becomes the best solution, outperforming all others on single-shot performance. If we know that we frequently target the same platform states then On-Demand-Cache has the potential to outperform No-Cache



(a) Logarithmic y axis, Target State Cache Precomputation times for Turn-on sequence.

(b) Logarithmic y axis, Target Platform State Calculation times for Turn-on sequence.



(c) Logarithmic y axis, Target Platform State Calculation times for Turn-on sequence. Zoomed-in to show Full-Cache outperforming other variants.

Figure 8.3: Cache Precomputation and Target Platform State Calculation times

for larger platforms.

Continuing with the next stage: Sequence generation, Figure 8.6a, and Figure 8.6b. Our implementations consistently outperform the baseline by several orders of magnitude and exhibit much better growth behaviour as the platform size increases.

There is also no variance between the same targets of our variants. Sequencing the platform to "turn-on" takes about the same amount of time as it does to sequence "turn-off". While sequence generation times could vary between approaches that somehow limited the number of conductors that have to change, in our evaluation the same conductors have to change no matter which target platform state or sequence is chosen, so the exact target state generated has little influence on sequencing, in addition to the fact that there is very little, if any, difference in the target platform states generated by our variants.

But qualitatively, these results are encouraging with regards to online-power management

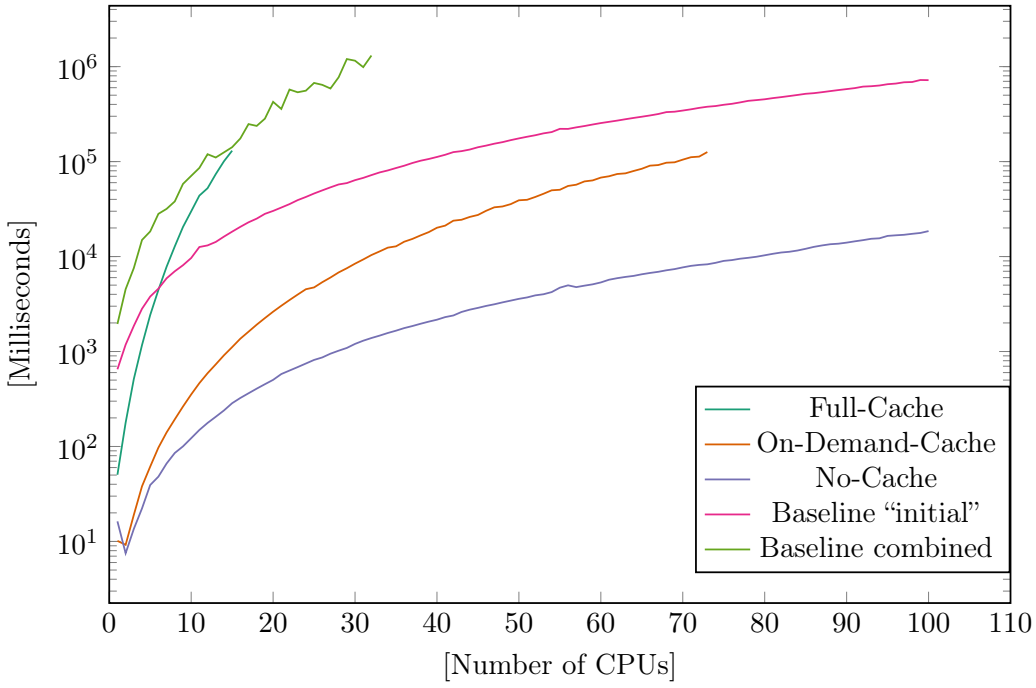


Figure 8.4: Logarithmic y axis, Non-baseline are showing Target Platform State Calculation + Target State Cache Precomputation for Turn-on sequence. Baseline is showing times for the two states necessary for both Turn-on and Turn-off sequences, as well as a combined value adding the two.

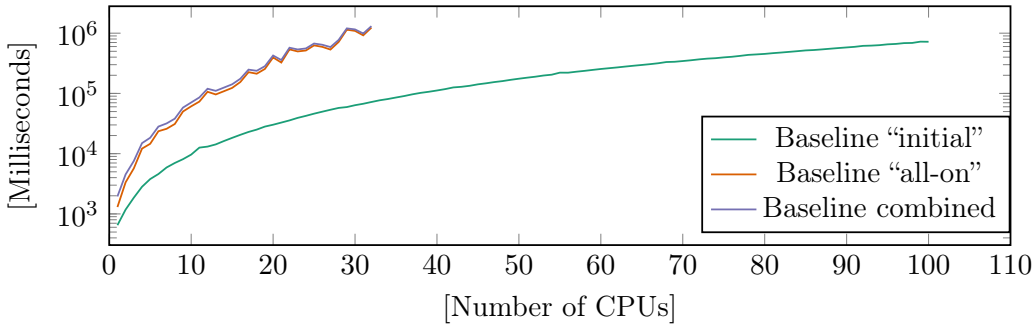


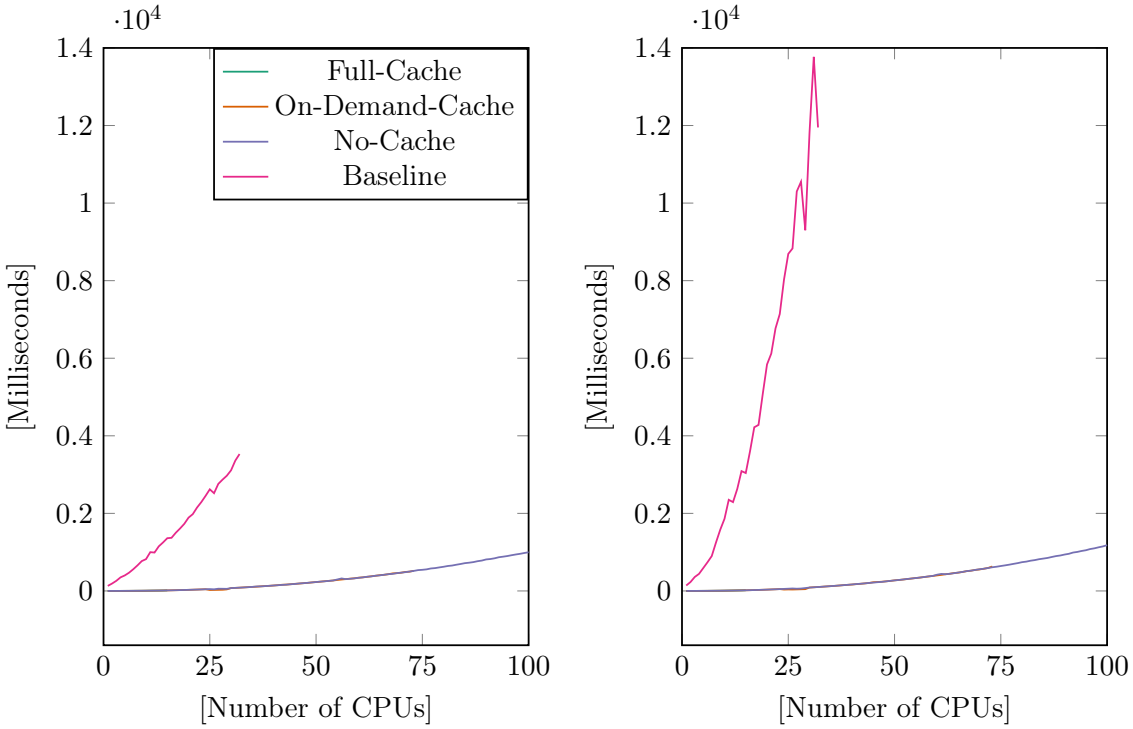
Figure 8.5: Logarithmic y axis, Baseline state computation for "init", "all-on" states and a combined measure for both added up. Compare Figure 8.4.

being feasible.

To confirm that our results on the Precision Desktop carry over to the BMC, we compare Target Platform State Calculation + Target State Cache Precomputation and Sequence Generation times for all three variants in Figure 8.7a, Figure 8.7b, Figure 8.7c, Figure 8.8a, Figure 8.8b and Figure 8.8c.

As expected, the Desktop significantly outperforms the BMC, but more importantly for both Target State and Sequence generation the BMC shows scaling behaviour matching roughly the one on the Desktop.

We are of course also interested in the absolute numbers for finding a target platform state and sequence on the BMC, and when we do we find that while for smaller platform sizes



(a) Sequence Generation times for Turn-on sequence

(b) Sequence Generation times for Turn-off sequence, see Figure 8.6a to the left for the Legend

Figure 8.6: Comparison showing Sequence Generation times for the Turn-on and Turn-off sequences respectively. Our implementations results very closely track each other, and that the baseline takes significantly longer to generate a sequence.

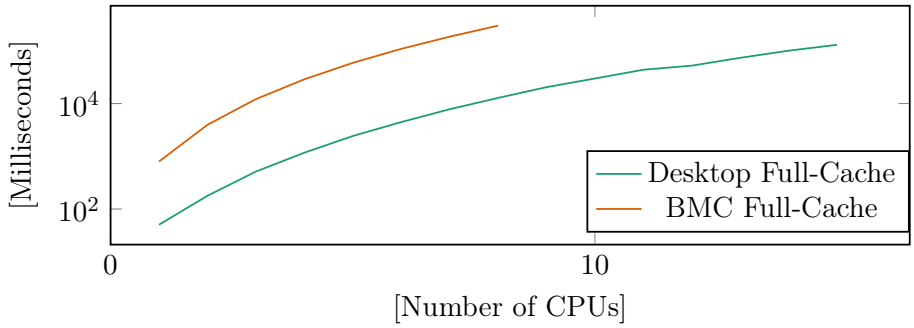
Full-Cache is feasible, for more than $3-4$ CPUs pre-generating the entire cache becomes far too expensive, and the near-zero precomputation efforts by the On-Demand-Cache and No-Cache variants become very appealing. However, Full-Cache, given a cache, still generates an actual target platform state faster than either of the other variants, saving 10s of milliseconds against No-Cache for small platforms.

If very low latency target platform state generation were desired for larger platforms, then as discussed previously, Full-Cache would have to become significantly smarter and not just pre-compute the entire cache. Perhaps critical target platform states could be generated first, or similar heuristics developed, but we leave this exploration to future work.

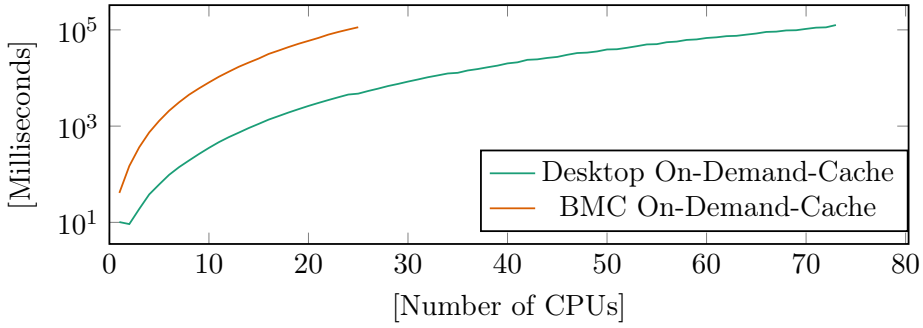
8.1.3 Summary

We were able to show that out target platform state and sequencing solutions from chapter [REFERENCE],[REFERENCE] outperform prior work from [3] by orders of magnitude, that they are suitable for online power management usage and able to scale to larger platform sizes.

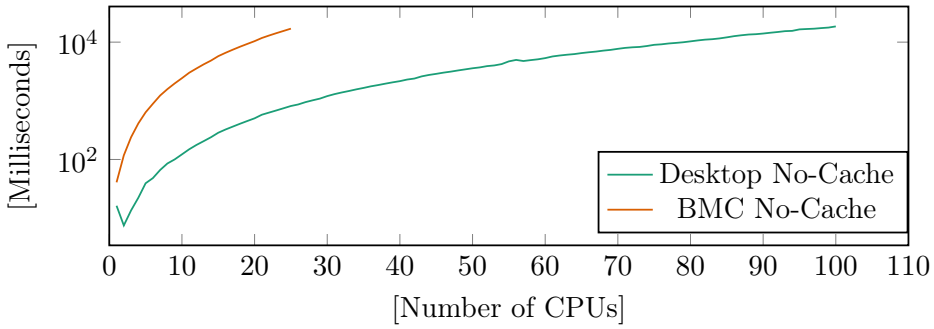
We verified this through running experiments both on a normal PC as well as an actual Enzian BMC, where for 2 virtual CPUs with about 4 seconds of pre-computation we can generate target platform states in about 46ms and sequences in about 6ms, for a total precomputed latency of roughly 52ms. If we generate the target platform states on-demand we can generate a target state in about 112ms, for a total worst-case latency of 118ms,



(a) Logarithmic y axis, Target Platform State Calculation + Target State Cache Precomputation for Full-Cache Turn-on sequences.



(b) Logarithmic y axis, Target Platform State Calculation + Target State Cache Precomputation for On-Demand-Cache Turn-on sequences.



(c) Logarithmic y axis, Target Platform State Calculation + Target State Cache Precomputation for No-Cache Turn-on sequences.

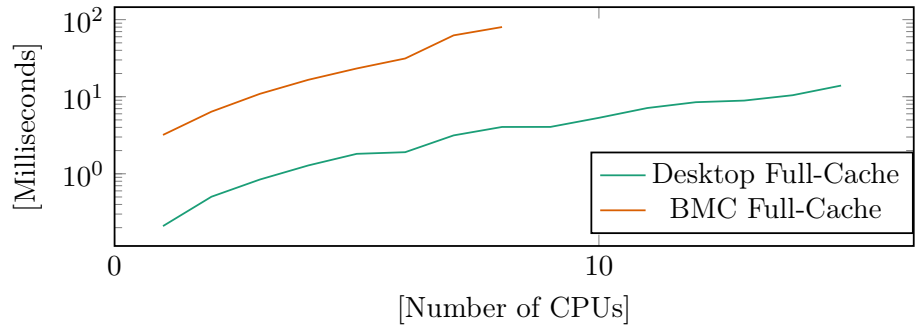
Figure 8.7: Comparison of Target Platform State Calculation + Target State Cache Precomputation times on the Zynq BMC and the Precision Dekstop

which we find to be fast enough for interactive usage.

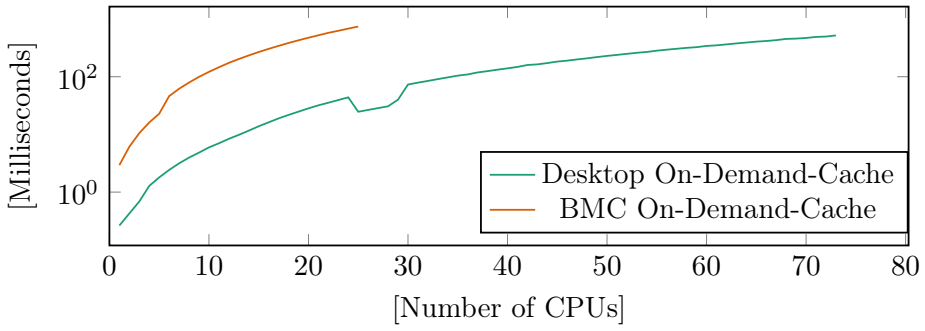
8.2 Simple Correctness of Sequencing and State Generation

Because our implementation is unfortunately not complete enough to allow a functional evaluation by actually executing a sequence on an Enzian, we can nonetheless attempt to show that our solution and implementation works based on a comparison to a solution that is known to generate working sequences.

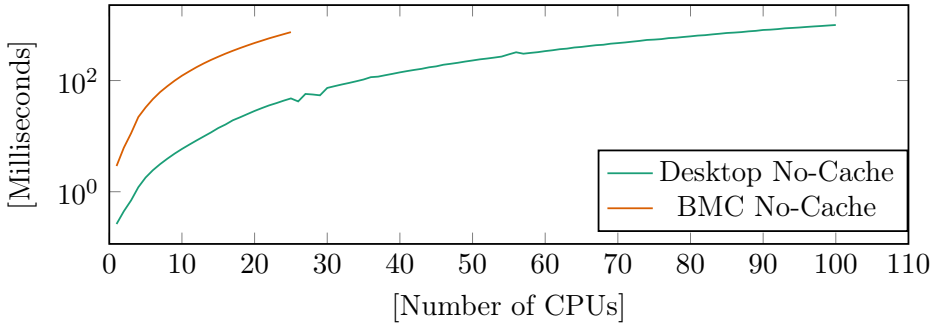
We argue that if a sequence graph generated by our implementation accepts the sequence generated by the implemetation in Knüsel [3] as a valid linear extension then that is at



(a) Logarithmic y axis, Sequence Generation for Full-Cache Turn-on sequences.



(b) Logarithmic y axis, Sequence Generation for On-Demand-Cache Turn-on sequences.



(c) Logarithmic y axis, Sequence Generation for No-Cache Turn-on sequences.

Figure 8.8: Comparison of Sequence Generation times on the Zynq BMC and the Precision Dekstop

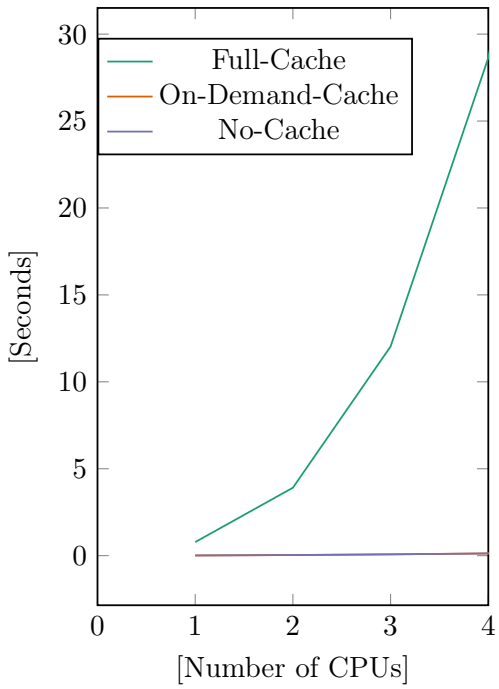
least an indication that our sequence graph is correct.

8.2.1 Setup

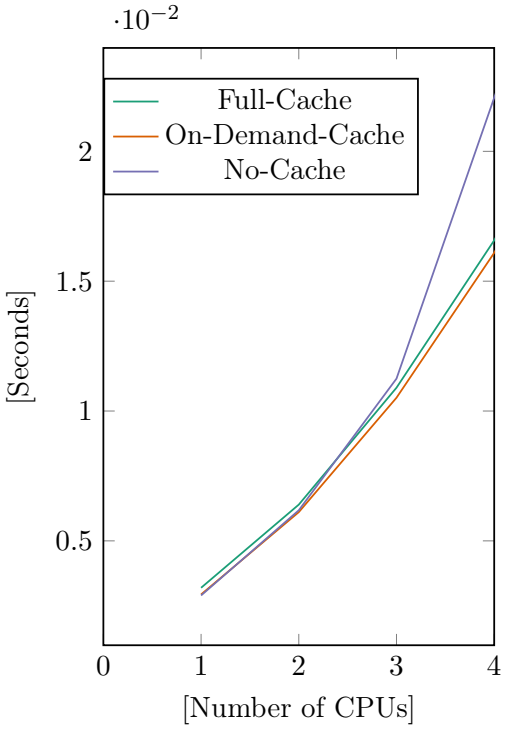
We generate a sequence with the basline from [3], and let our implementation generate a schedule graph. We use the same platform specification generation procedure as in Section 8.1 and generate a platform spec with a single ThunderX-CPU.

Then, we sanitize the sequence generated by the basline, removing unnecessary details like ports or value ranges.

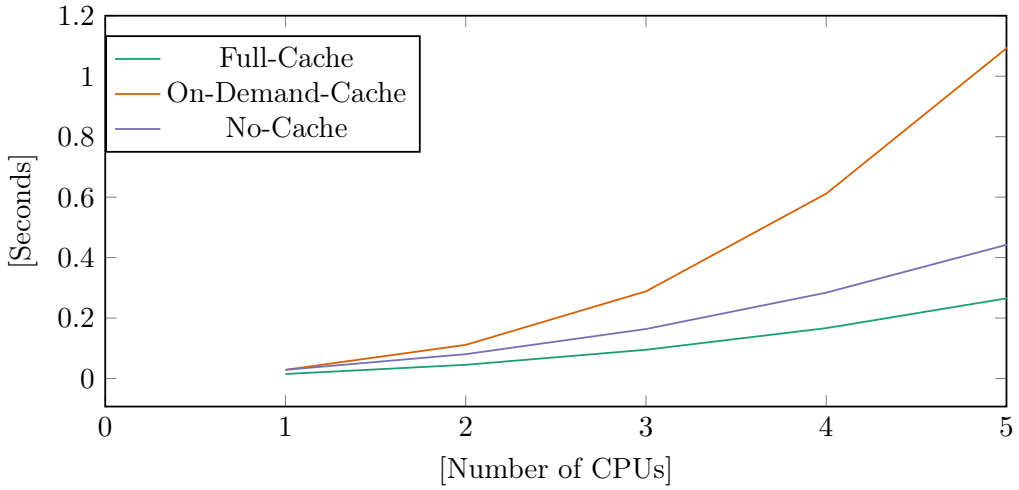
Note that the baseline schedule outputs commands, for example setting the warning levels for the ISPPAC component monitors, that our implementation would do on-the-fly and not explicitly encode in the sequence. The baseline also "configures" components that do not have an explicit "Configured" state. We consider the "Configured" and "Powered" states



(a) Logarithmic y axis, Target State Cache Precomputation for Turn-on sequences on the BMC.



(b) Logarithmic y axis, Sequence Generation for Turn-on sequences on the BMC.



(c) Logarithmic y axis, Target Platform State Calculation for Turn-on sequences on the BMC.

Figure 8.9: Comparisons of Total Target Platform State Calculation and Sequence Generation on the BMC for all variants, zoomed in to small platform sizes. We see that for effort for the target state cache increases very quickly on the BMC, but also that calculating the actual target platform state is always faster than all other variants. For larger platforms, full state precomputation is obviously not feasible on the BMC, but a smarter approach where the cache is partially generated on-demand and in the background, or adding heuristics for which target states to generate target platform statest first, could preserve the advantages of having a cache, while thinning the pain of having to compute it.

equivalent for those components, or "On" if they have no "Powered" state.

```
sequence: (
 (1
  set-to
  wire.b-psup-on)
 (1
  configure
  pac-cpu-0
 (1.5
  monitor
  wire.psup-pgood
 (1.5
  monitor
  wire.3v3-psup
 (1.5
  monitor
  wire.5v-psup
 (1.5
  monitor
  wire.12v-cpu0-psup
 (2 configure ina226-ddr-cpu-13-0)
 (2 configure si5395-clk-main)
 [...]
)
```

Listing 1: Illustrative subset of the "sanitized" baseline output.

We then translate it using the following equivalence:

- `(i set-to wire.X)` and *X_permitted*.
- `(i configure X)` and *X:->Configured*.
- `(i monitor X)` and *X_happened*.
- `(n wait (x ms))` and *component_wait*.

We use the names from the baseline sequence and the step names as found in the debug sequence graph output (which outputs the internal names of the sequence steps).

Finally, we go through the baseline sequence steps one after the other and ensure after every step that the sequence graph our implementation generated considers the step valid, given the steps alreay taken.

For every baseline sequence step we check that no earlier sequence step is a parent in the sequence graph.

See Figure A.1, Listing 3, and Listing 4 in the appendix for the full sequence graph and baseline outputs that were used.

8.2.2 Result

We find that the sequence graph our implementation generates accepts the sequence with the baseline from [3] as valid, with the following deviations:

```
b-psup-on_permitted

pac-cpu-0:->Configured

psup-pgood_happened
3v3-psup_happened
5v-psup_happened
12v-cpu0-psup_happened

ina226-ddr-cpu-13-0:->Configured

[...]

cpu_1_wait

[...]
```

Listing 2: Illustrative subset of the translated sanitized baseline output.

1. The baseline sequence checks if the *clk-main* conductor has changed later than our solution would, after telling the regulator *si5395-clk-cpu-0* to configure itself. This is due to a minor difference in the platform specifications used, where the baseline forces the *SI5395 CLK_IN* input to 0 while powered, while we allow it to float between 0 and 50 millihertz. We do not consider this a deviation that indicates that our implementation generates sequences incorrectly.

8.2.3 Interpretation

We interpret this result as positive indication that our platform target state and sequence generation mechanisms output correct sequences, but acknowledge that actually executing the sequence on an Enzian BMC and observing the platform taking the desired state correctly remains outstanding.

8.3 Summary

We verified our target platform state and sequence generation runtimes by comparing them against a baseline form [3], and also executing them on an actual Enzian Zynq BMC. We also provide strong indication that our target platform state and sequence generation works correctly by verifying the baseline, which is known to generate correct sequences, against a sequence graph we generated.

We have succesfully shown that out target platform state generation and sequencing implementation is able to generate sequences quickly enough for online, dyamic usage, and that it likely does so correctly.

Chapter 9

Conclusion

We conclude the thesis with a list of improvements we think can be made on this work, as well as avenues for future work only partially related to the solution presented in this thesis that we encountered, and a concluding statement.

9.1 Future Work

> "Prima facie evidence suggests that there could be a case for further investigation, to establish whether or not enquiries should be put in hand. [...] Nevertheless, it should be stressed that available information is limited and relevant facts could be difficult to establish with any degree of certainty."
>
> Sir Nigel Hawthorne as Sir Humphrey Appleby – Yes, Minister

There are quite a few improvements that we could make that have built up along the way. The following is a selection of future improvements and directions for research, organized roughly by "area".

General improvements:

- Configuration management: The way the configuration management is currently solved and the way it interacts with the rest of the system feels "clunky". This subsystem needs improvement, possibly by allowing/requiring the component-DES to keep additional state internally.
- FPGA offloading of target platform state and/or sequence generation as CSP: The Enzian BMC is an SOC with both a CPU and an FPGA. Instead of arduously generating the target platform state and sequences on the CPU, it may be possible to offload a CSP instance or similar encoding of either problem to the FPGA.
- Error margins: A regression over Knüsel [3] are error margins for ports, assignments and measurements. Because this increases the complexity for otherwise very simple operations ("can this assignment fulfill this measurement", "does this output value fit this state") this was left out, but should definitely find its way back into the solution.

- Remove restrictions: A catch-all improvement. Removing restrictions we place on the platform could allow for more platforms to be modelled by our solution.
- Explicit support for more faults: Currently the platform is primarily concerned with voltages and clock-frequencies. More explicit support for the other kinds of faults, relating to current, temperature etc., is needed as they are almost, but not quite the same as voltages. There are also completely faults like the fans ceasing operation suddenly that we do not explicitly support everywhere. Fundamentally, the component-DES can deal with these with minor changes, but the rest of the platform needs to be changed slightly to accomodate these new fault types.

Sequencing/State generation:

- Compilation of full Schult/Knuselian problem with online guarantees: While dismissed during this thesis, it may still be possible to "compile" the more general Schult or Knüsel models and ensure low latencies that way.
- Other multi-transition sequencing approaches: We feel like our multi-transition sequencing should work, but also feels inelegant. Knowing full well of the increase in complexity, an integrated multi-transition sequencing solution that inherently understands transitory states would make for a very interesting solution.
- Sequence Repair: Discussed only very briefly, if we could repair our sequences instead of having to throw them away at the slightest change in model state, we could save a lot of expensive re-computation effort.
- Target state cache repair: Pre-computing the target cache is the single most expensive action our solution has to take. Instead of throwing the whole cache away when a component-DES requests it, it may be possible to retain a lot of the state that went into generating it and re-generate based on that instead of starting again from scratch.
- Target State Computation: Using a maximum independent set as the basis for our target state computation is theoretically elegant, but we have a suspicion that there may be a much simpler solution akin to our sequencing that generates a target platform state iteratively and faster.
- Smarter State Cache: Though dependent on the individual requirements of the user and hardware, implementing different state cache computation methods and allowing the user to choose from among them, maybe even specify them somehow, would allow for better adaptability to different hardware environments and make it easier to evaluate them in-depth.

Controller:

- Allow multiple outstanding actions: We currently limit the number of actions that our model has initiated at any point in time to one, but schedules could be executed on the platform a lot faster if multiple Read/Write/Configures could be outstanding at any time. Maybe having either multiple reads or a single "change" action at any point in time is fine under certain circumstances, as long as the reads are unable to override each other.
- Let component-DES request high-priority actions: This would help with fault reaction latency. If a component-DES requests a new target state, to mitigate a fault, then there is currently no way for that request to be prioritized, wasting time. However,

it is unclear which other *normal*-priority actions are also implicitly high-priority, because the component-DES action may depend on them.

- Component-DES or Transition Manager timeout: The component-DES or transition manager cannot currently time out, meaning that WaitForVoltage or similar behaviours are not possible. The possible solutions we considered all either involved taking control away from the component-DES, which we consider an impure solution because the control logic for a component should be concentrated in its DES, or extending the DES either with some internal clock, or a model-clock that informs the DES when a request has timed out.
- Partial-consistency Model: A Partial-consistency model is a model that does not require every component on the platform to be fully satisfied with its state. This would allows for immediate execution of actions that don't need full-model visibility, like the components in the two separete power trees for the CPU and FPGA on an Enzian, but would be complicated to implement, as it would require a rework of large parts of the model.

9.2 Summary

In this thesis, we have invetigated declarative dynamic power management. In pursuit of this goal, we first investigated three general approaches to the problem at hand, before deciding that none of them were sufficient for our online, dyanmic target use case and creating our own solution, inspired by the restrictions introduced by Knüsel [3].

We came to the conclusion that appropriately dealing with the vast heterogeneity of component behaviours that we want to manage requires a solution that allows components to "manage themselves", treating them as a black-box discrete event system that we make output abstract state information that we can use to generate target platform states and sequences.

We introduce a new way of generating target platform states by viewing it as a MIS-based optimization problem. Similarly, we come up with a novel way of genrating a complete sequence between two fully specified target platform states by building a DAG from state assignments and inter-state transitions. Our solution also introduces a way to generate composite sequences with multiple changes per conductor by generating a least-intermediate target state that we have high confidence can be sequenced to.

We evaluate our target platform state and sequence generation mechanisms' performance against a baseline [3], and on an actual Enzian BMC. We also demonstrate that we can generate sequence graphs that closely match Knüsel [3] sequences.

The complete design, while unimplemented, fulfills the requirements for dynamically managing an Enzian, bringing it into stable target states, and reacting to unexpected faults safely.

Bibliography

[1] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An open, general, cpu/fpga platform for systems software research," ACM, Feb. 2022, pp. 434–451, ISBN: 9781450392051. DOI: 10.1145/3503222.3507742. [Online]. Available: https://dl.acm.org/doi/10.1145/3503222.3507742.

[2] J. Schult, "A model-based approach to platform-level power and clock management," 2020. DOI: 10.3929/ETHZ-B-000490632. [Online]. Available: https://doi.org/10.3929/ethz-b-000490632.

[3] M. Knüsel, "Optimizing declarative power sequencing," 2021. DOI: 10.3929/ETHZ-B-000533011. [Online]. Available: https://doi.org/10.3929/ethz-b-000533011.

[4] C. Allardice, E. R. Trapnell, E. Fermi, L. Fermi, and R. C. Williams, "The first reactor [40th anniversary commemorative edition]," Office of Scientific and Technical Information, Dec. 1982. DOI: 10.2172/782931. [Online]. Available: http://www.osti.gov/servlets/purl/782931-KfJR9N/webviewable/.

[5] "Virtual media vulnerability in bmc opens servers to remote attack," [Online]. Available: https://github.com/eclypsium/USBAnywhere.

[6] C. Heimhofer, "Towards high-assurance board management controller software," 2021. DOI: 10.3929/ETHZ-B-000490635. [Online]. Available: https://doi.org/10.3929/ethz-b-000490635.

[7] L. Benini and G. D. Micheli, *Dynamic Power Management*. Springer US, 1998. DOI: 10.1007/978-1-4615-5455-4.

[8] *I2c-bus specification and user manual*, Oct. 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[9] L. Humbel, D. Schwyn, N. Hossle, R. Haecki, M. Licciardello, J. Schaer, D. Cock, M. Giardino, and T. Roscoe, *A model-checked i2c specification*, Aug. 2021. DOI: 10.1007/978-3-030-84629-9_10. [Online]. Available: https://link.springer.com/10.1007/978-3-030-84629-9_10.

[10] *System management bus (smbus) specification*, Mar. 2018. [Online]. Available: http://smbus.org/specs/SMBus_3_1_20180319.pdf.

[11] *Pmbus™ power system management protocol specification*. [Online]. Available: https://pmbus.org/current-specifications/.

[12] J. Schult, D. Schwyn, M. Giardino, D. Cock, R. Achermann, and T. Roscoe, "Declarative power sequencing," *ACM Transactions on Embedded Computing Systems*, vol. 20, pp. 1–21, 5s Oct. 2021, ISSN: 1539-9087. DOI: 10.1145/3477039. [Online]. Available: https://dl.acm.org/doi/10.1145/3477039.

[13] R. Weigel and B. Faltings, "Compiling constraint satisfaction problems," *Artificial Intelligence*, vol. 115, pp. 257–287, 2 Dec. 1999, ISSN: 00043702. DOI: 10.1016/

S0004-3702(99)00077-6. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0004370299000776.

[14] H. Fargier, F. Maris, and V. Roger, "Temporal constraint satisfaction problems and difference decision diagrams: A compilation map," vol. 2016-January, IEEE, Nov. 2015, pp. 429–436, ISBN: 978-1-5090-0163-7. DOI: 10.1109/ICTAI.2015.71. [Online]. Available: http://ieeexplore.ieee.org/document/7372167/.

[15] M. Balunović, P. Bielik, and M. Vechev, "Learning to solve smt formulas," S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/68331ff0427b551b68e911eebe35233b-Paper.pdf.

[16] H. Turner, *Polynomial-length planning spans the polynomial hierarchy*, 2002. DOI: 10.1007/3-540-45757-7_10. [Online]. Available: http://link.springer.com/10.1007/3-540-45757-7_10.

[17] B. Bonet, R. Fuentetaja, Y. E-Martín, and D. Borrajo, "Guarantees for sound abstractions for generalized planning," AAAI Press, 2019, pp. 1566–1573, ISBN: 9780999241141. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3367243.3367256.

[18] S. Jiménez, J. Segovia-Aguas, and A. Jonsson, "A review of generalized planning," *The Knowledge Engineering Review*, vol. 34, e5, Mar. 2019, ISSN: 0269-8889. DOI: 10.1017/S0269888918000231. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0269888918000231/type/journal_article.

[19] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, C. G. Cassandras and S. Lafortune, Eds. Springer US, 2008, pp. 1–771, ISBN: 978-0-387-33332-8. DOI: 10.1007/978-0-387-68612-7. [Online]. Available: http://link.springer.com/10.1007/978-0-387-68612-7.

[20] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Diagnosability of discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 40, pp. 1555–1575, 9 1995, ISSN: 00189286. DOI: 10.1109/9.412626. [Online]. Available: http://ieeexplore.ieee.org/document/412626/.

[21] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, 4 Apr. 1989, ISSN: 00189219. DOI: 10.1109/5.24143. [Online]. Available: http://ieeexplore.ieee.org/document/24143/.

[22] G. Stremersch, *Supervision of Petri Nets*. Springer US, 2001, ISBN: 978-1-4613-5603-5. DOI: 10.1007/978-1-4615-1537-1. [Online]. Available: http://link.springer.com/10.1007/978-1-4615-1537-1.

[23] C. A. Petri, "Kommunikation mit automaten," Technische Hochschule Darmstadt, Jun. 1962. [Online]. Available: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/.

[24] V. Valberg and R. Davidrajuh, "Estimating salmon price rise due to the increased presence of lice caused by global warming: A petri net based approach," *International journal of simulation: systems, science & technology*, Apr. 2021, ISSN: 1473-804X. DOI: 10.5013/IJSSST.a.22.01.05. [Online]. Available: https://edas.info/doi/10.5013/IJSSST.a.22.01.05.

[25] M. Zhou and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Springer US, 1993, ISBN: 978-1-4613-6368-2. DOI: 10.1007/978-1-4615-3126-5. [Online]. Available: http://link.springer.com/10.1007/978-1-4615-3126-5.

[26] F. G. Commoner, *Deadlocks in Petri-nets, CA / Massachusetts Computer Associates, Inc.* Massachusetts Computer Assoc., Inc.; 1972, vol. 7206-2311, Frederic G.

Commoner: Deadlocks in Petri Nets. Applied Data Research Inc., Wake-
152 field, Massachusetts 01880. Report Nr. CA-7206-2311 (1972). [Online]. Available: https://www.tib.eu/de/suchen/id/TIBKAT%3A492829667.

[27] K. Lautenbach, *Liveness in Petri Nets, Interner Bericht / Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn, Institut für Informationssystemforschung ISF*. Selbstverl. GMD; 1975, vol. 75,2. [Online]. Available: https://www.tib.eu/de/suchen/id/TIBKAT%3A017519519.

[28] T. Agerwala and M. Flynn, "Comments on capabilities, limitations and "correctness" of petri nets," *ACM SIGARCH Computer Architecture News*, vol. 2, pp. 81–86, 4 Dec. 1973, ISSN: 0163-5964. DOI: 10.1145/633642.803973. [Online]. Available: https://dl.acm.org/doi/10.1145/633642.803973.

[29] T. Agerwala, "Complete model for representing the coordination of asynchronous processes," Technical Information Center, Jul. 1974. DOI: 10.2172/4242290. [Online]. Available: http://www.osti.gov/servlets/purl/4242290/.

[30] M. Hack, "Petri net languages," *MIT Computation Structures Group Memo*, 124 Jul. 1975.

[31] E. Badouel, L. Bernardinello, and P. Darondeau, *Petri Net Synthesis*. Springer Berlin Heidelberg, 2015, ISBN: 978-3-662-47966-7. DOI: 10.1007/978-3-662-47967-4. [Online]. Available: http://link.springer.com/10.1007/978-3-662-47967-4.

[32] A. Ehrenfeucht and G. Rozenberg, "Partial (set) 2-structures - part i: Basic notions and the representation problem," *Acta Informatica*, vol. 27, pp. 315–342, 4 Mar. 1990, ISSN: 00015903. DOI: 10.1007/BF00264611.

[33] ——, "Partial (set) 2-structures - part ii: State spaces of concurrent systems," *Acta Informatica*, vol. 27, pp. 343–368, 4 Mar. 1990, ISSN: 00015903. DOI: 10.1007/BF00264612.

[34] E. Badouel and P. Darondeau, *Theory of regions*, 1998. DOI: 10.1007/3-540-65306-6_22. [Online]. Available: http://link.springer.com/10.1007/3-540-65306-6_22.

[35] A. Ghaffari, N. Rezg, and X. Xie, "Design of a live and maximally permissive petri net controller using the theory of regions," *IEEE Transactions on Robotics and Automation*, vol. 19, pp. 137–142, 1 Feb. 2003, ISSN: 1042-296X. DOI: 10.1109/TRA.2002.807555. [Online]. Available: http://ieeexplore.ieee.org/document/1177171/.

[36] R. Lorenz, S. Mauser, and R. Bergenthum, "Theory of regions for the synthesis of inhibitor nets from scenarios," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4546 LNCS, pp. 342–361, 2007, ISSN: 16113349. DOI: 10.1007/978-3-540-73094-1_21. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-73094-1_21.

[37] F. Basile, "Overview of fault diagnosis methods based on petri net models," IEEE, Jun. 2014, pp. 2636–2642, ISBN: 978-3-9524269-1-3. DOI: 10.1109/ECC.2014.6862631. [Online]. Available: http://ieeexplore.ieee.org/document/6862631/.

[38] *Github - rust-lang/nomicon: The dark arts of advanced and unsafe rust programming.* [Online]. Available: https://github.com/rust-lang/nomicon.

[39] E. Szpilrajn, "Sur l'extension de l'ordre partiel," *Fundamenta Mathematicae*, vol. 16, pp. 386–389, 1930, ISSN: 0016-2736. DOI: 10.4064/fm-16-1-386-389. [Online]. Available: http://www.impan.pl/get/doi/10.4064/fm-16-1-386-389.

[40] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM Journal on Computing*, vol. 15, pp. 1036–1053, 4 Nov. 1986, ISSN: 0097-5397.

DOI: 10.1137/0215074. [Online]. Available: http://epubs.siam.org/doi/10.1137/0215074.

Appendix A

Evaluation Artifacts

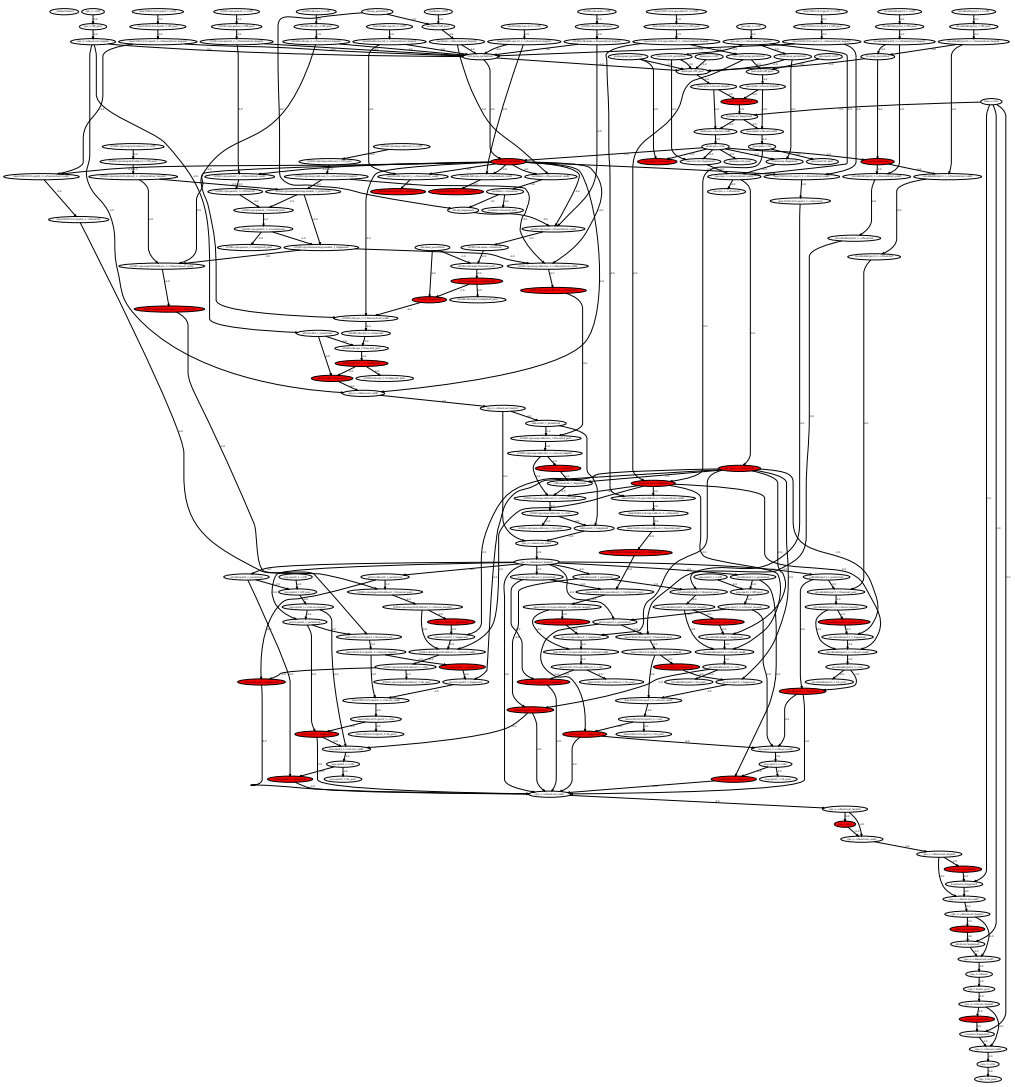


Figure A.1: Marked sequence graph for a single ThunderX CPU used in the evaluation. The red vertices are the steps from the baseline output, see Listing 4.

```
sequence: (
```

```
(1
 set-to
 wire.b-psup-on)
(1
 configure
 pac-cpu-0
(1.5
 monitor
 wire.psup-pgood
(1.5
 monitor
 wire.3v3-psup
(1.5
 monitor
 wire.5v-psup
(1.5
 monitor
 wire.12v-cpu0-psup
(2 configure ina226-ddr-cpu-13-0)
(2 configure si5395-clk-main)
(2 configure ina226-ddr-cpu-24-0)
(2 configure si5395-clk-cpu-0)
(2
 configure
 isl-vdd-ddrcpu13-0
(2
 configure
 isl-vdd-ddrcpu24-0
(2 configure ir3581-cpu-parent-0)
(2.5
 monitor
 wire.clk-main
(2.5
 monitor
 wire.pll-ref-clk0-0
(3
 configure
 ir3581-cpu-loop-vdd-core-0
(3
 configure
 ir3581-cpu-loop-0v9-vdd-oct-0
(4
 set-to
 wire.vdd-core0-en-0)
(4
 configure
 max15301-1v5-cpu-vdd-oct-0
(5
 set-to
```

```
 wire.vdd-oct-en0-l2-0)
(5
 set-to
 wire.en-1v5-cpu-vdd-oct-0)
(5.5
 monitor
 wire.w0v9-vdd-oct0-0
(5.5
 monitor
 wire.w1v5-cpu-vdd-oct-0
(6
 set-to
 wire.en-2v5-cpu13-0)
(6
 set-to
 wire.en-2v5-cpu24-0)
(6
 set-to
 wire.en-vdd-ddrcpu13-0)
(6
 set-to
 wire.en-vdd-ddrcpu24-0)
(6.5
 monitor
 wire.vdd-ddrcpu13-0
(6.5
 monitor
 wire.vdd-ddrcpu24-0
(6.5
 monitor
 wire.w2v5-cpu13-0
(6.5
 monitor
 wire.vtt-ddrcpu13-0
(6.5
 monitor
 wire.w2v5-cpu24-0
(6.5
 monitor
 wire.vtt-ddrcpu24-0
(7 wait (3 ms))
(8
 set-to
 wire.b-spi-sel-n)
(9
 set-to
 wire.pll-dc-ok)
(10
 set-to
```

```
  wire.c-reset-n)
)
```

Listing 3: Sanitized baseline output.

```
b-psup-on_permitted

pac-cpu-0:->Configured

psup-pgood_happened
3v3-psup_happened
5v-psup_happened
12v-cpu0-psup_happened

ina226-ddr-cpu-13-0:->Configured
si5395-clk-main:->Configured
ina226-ddr-cpu-24-0:->Configured
si5395-clk-cpu-0:->Configured
isl-vdd-ddrcpu13-0:->Configured
isl-vdd-ddrcpu24-0:->Configured
ir3581-cpu-parent-0:->Configured

clk-main_happened
pll-ref-clk0-0_happened

ir3581-cpu-loop-vdd-core-0:->Configured
ir3581-cpu-loop-0v9-vdd-oct-0:->Configured

vdd-core0-en-0_permitted

max15301-1v5-cpu-vdd-oct-0:->Configured

vdd-oct-en0-l2-0_permitted
en-1v5-cpu-vdd-oct-0_permitted

w0v9-vdd-oct0-0_happened
w1v5-cpu-vdd-oct-0_happened

en-2v5-cpu13-0_permitted
en-2v5-cpu24-0_permitted
en-vdd-ddrcpu13-0_permitted
en-vdd-ddrcpu24-0_permitted

vdd-ddrcpu13-0_happened
vdd-ddrcpu24-0_happened
w2v5-cpu13-0_happened
vtt-ddrcpu13-0_happened
w2v5-cpu24-0_happened
vtt-ddrcpu24-0_happened

cpu_1_wait
```

```
b-spi-sel-n_permitted
pll-dc-ok_permitted
c-reset-n_permitted
```

Listing 4: Translated sanitized baseline output.

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):

Declarative Dynamic Power Management

Authored by (in block letters):

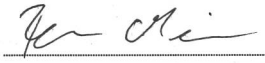
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Meier	Roman

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)**
Zürich, 21.09.2022	

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