Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 290b

Systems Group, Department of Computer Science, ETH Zurich

A model-based approach to platform-level power and clock management

by

Jasmin Schult

Supervised by

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

February 2020 - August 2020

inf | Informatik
Computer Science

Abstract

The correct management of a platform's power and clock resources is vital for the correct operation of the platform's hardware, which in turn forms the cornerstone of any assurances provided by software. In light of this, the firmware implementing such a power and clock management solution should ideally provide strong correctness guarantees. However, to the extent of our knowledge, the current state of the art does not go beyond manually-coded point solutions.

This thesis approaches the problem of power and clock management in a more principled fashion: It proposes a model that captures the behaviour of the platform in this management context. Additionally, a set of mechanisms are presented that, if applied to any model instance, generate management solutions for the platform described by that instance. If the correctness of these mechanisms were formally verified, which was not possible within the time constraints of this thesis, provably correct management solutions could be generated for any platform whose behaviour can be expressed by the model.

Acknowledgements

I would like to extend my heartfelt thanks to Daniel Schwyn, Dr. David Cock, Dr. Michael Joseph Giardino and Prof. Dr. Timothy Roscoe. Without them, this thesis would not have been possible, not only because they developed the fundamental idea behind it, but also because their support, suggestions and feedback was invaluable.

In light of this, it felt more natural to me to write the entirety of this thesis (with the exception of these acknowledgements) in the first-person-plural point of view. However, I do take full responsibility for any kind of mistakes or lack of clarity in the argumentations or descriptions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To some extent, every piece of software relies on the underlying hardware platform to function correctly. Even formally verified software is no exception; as noted by Klein et al in the discussion of their verification of seL4 ([15], p. 29):

> The best validated formal theorem will not guarantee correct behaviour if processor and memory are melting underneath.

There are several ways in which the correct functionality of hardware can be undermined. For instance, by applying an inappropriate supply voltage or a too skewed clock signal that violates the circuitry's timing constraints. Consequently, the *correct* management of the platform's power and clock resources constitutes a vital precondition for any software guarantees to hold.

This thesis attempts to make a step towards providing stronger guarantees for the correctness of this management. Before we concretise this goal, we discuss the current state of the art of power and clock management.

1.1 State of the Art

On modern computing platforms, power and clock management is not a trivial matter. A CPU might for instance require an array of different supply voltages that must come online in specific sequences during the bootstrap process. Therefore, this management along with other general platform maintenance functionality is handed off to a dedicated microcontroller. Depending on vendor and platform, this microcontroller is known under many names, for example *Management Engine* in the case of Intel. In this thesis, we will use the more general term *Baseboard Management Controller* (BMC).

The general state of the firmware running on BMCs, as far as disclosed, is at odds with the high level of privilege with which it executes [6]. This is also includes the parts dedicated to power and clock management: To our knowledge, the current published state of the art does not go beyond manually-coded point solutions and no attempts at formal verification have been made.

1.2 Goal of this thesis

In this thesis, we will approach the problem of power and clock management in a fashion that separates policy from mechanism: We devise a model that captures *correct* and relevant platform behaviour with respect to power and clock management. On top of this model, we construct a set of mechanisms that generate the BMC actions necessary to correctly manage the platform described by any model instance.

Our approach is promising from the point of view of formal verification: If we prove the correctness of the mechanisms with respect to the intended model semantics, we can generate verified management code for any platform our model can capture. Within the time constraints of this thesis, we were not able to completely formalise our model semantics and perform this correctness proof, yet we are convinced that it would be possible.

We conclude this section by outlining the structure of our thesis and the purpose every chapter serves with respect to our management approach:

In chapter 3, we attempt to describe power and clock management as well as the involved components from a point of view that is as universal as possible. Since we have found almost no suitable literature to rely on, we complete this description with observations we have made in context of the Enzian research platform, which we will introduce in chapter 2, along with other scientific work that relates to this thesis.

Based on these general descriptions, we identify a subset of power and clock management our thesis will focus on in chapter 4. We then proceed to concretise the requirements our model and mechanisms must fulfil.

Chapter 5 presents the platform model we have devised, motivated by our earlier general description of power and clock management.

In chapter 6, we concretise the problems our mechanisms must solve and discuss their computational complexity. We then present an algorithm capable of providing the desired solution.

As a proof of concept, we have modelled the aforementioned Enzian platform. In chapter 7, we present some observations we have made in that process, as well as some general modelling advice.

Chapter 8 is dedicated to the implementation of our mechanisms as well as the model syntax.

We evaluate the performance of our mechanisms and perform a comparison to the management solution implemented for the Enzian platform in chapter 9.

Finally, we conclude our thesis and propose future work in chapter 10.

Chapter 2

Related Work

This chapter discusses the scientific work relevant for this thesis. After an overview of general related work, we will introduce two current projects of the Systems Group at ETH that have shaped the idea for a policy-mechanism separated approach to power and clock management.

2.1 General related work

We have structured this section into several subsections according to the discussed topic.

2.1.1 Power Management

The topic of optimally trading off performance and power consumption has been thoroughly studied. In scientific literature [3, 4], the process of deciding *when* a CPU or similar computational agent should be transitioning to a more or less performant power state is referred to as *dynamic power management*. This does not correspond to the power management problem we address in this thesis; we are concerned with *platform-level* power and clock management: The firmware running on the BMC must realise the aforementioned transitions between power states but does not make the decision of *when* to perform them.

2.1.2 BMC firmware

As mentioned in the introduction, the current state of the firmware running on BMCs is not satisfactory. In recent years, the disclosure of multiple resulting security vulnerabilities has drawn a lot of attention to this issue. As a consequence, first actions have been taken to improve the situation, of which we provide a non-comprehensive, high-level overview:

Up to a few years ago, BMC firmware was proprietary and not disclosed to the public. Since then, projects such as OpenBMC and u-bmc have disclosed some implementations to the public, with the aim of providing more transparency and in the hopes of collectively finding and fixing bugs and vulnerabilities. [9]

To ease data centre management, modern BMCs offer an array of remote management capabilities. To be able to do so, the firmware must implement some minimal OS functionality such as NIC drivers and network protocols. According to Narayanan et al, the resulting inherent complexity of the firmware code routinely introduces bugs and vulnerabilities. To address this issue, they propose the verified minimal OS *RedLeaf* in [17], that is "aimed at the needs of a diverse family of firmware subsystems".

In 2018, the National Institute of Standards and Technology (NIST) of the United States has released a set of guidelines to improve the protection as well as the detection and subsequent recovery of platform firmware from malicious attacks. [19]

2.1.3 Related Problems

As mentioned in the introduction, we aim to devise a model that captures *correct* platform behaviour. Quite generally, this can be achieved by placing appropriate *constraints* on said behaviour. The mechanisms we construct to extract management actions from our model therefore correspond to *Constraint Satisfaction Problems* (CSPs).

CSPs are a very broad class of problems. In the resulting wealth of scientific papers about these problems, one could almost certainly find a CSP formulation for a different problem that is very similar in character to the platform model we have devised. Thematically, we have found the most similar CSP instances to be concerned with the so-called *unit commitment problem* (see, for instance, [26]). Said is concerned with determining the optimal start-up and shut-down times of generating units in connection with electrical power production. However, these problem instances do not exhibit much structural similarity to our platform model.

2.2 Enzian

Until recently, commercial off-the-shelf hardware (CTOS) was ubiquitously used, from every-day to commercial usage. Nowadays, there is an increasing trend towards highly specialised or even custom hardware for commercial applications. This is problematic for academic systems software research: research on CTOS hardware does not generalise to custom hardware whereas custom hardware - if even disclosed - features such a high degree of specialisation that research is likely reduced to confirming that it works well for the intended use case but badly for others.[25]

This is where Enzian comes into play, a research platform built from scratch by the Systems Group at ETH. Its most extraordinary feature is the tight coupling of a server-class ThunderX CPU with a high-end FPGA. This coupling, combined with the configurability of the FPGA offers a tremendous amount of flexibility and can therefore be used to explore and simulate a lot of scenarios interesting to systems software research.

With its high-end CPU and FPGA, power and clock management on the Enzian platform is inherently non-trivial, and it thus also features a BMC. With the Enzian being designed from scratch, its complete platform layout and the functionality of every integrated circuit

are revealed. This offers a unique opportunity to reason about and explore the power and clock management actions the Enzian BMC must perform.

2.3 Sockeye

Sockeye is another on-going research project of ETH's Systems Group. Originally, *Sockeye* referred to the domain specific language developed by Daniel Schwyn in his master's thesis [22] to express the *address decoding net* model proposed by [2]. Meanwhile, Sockeye has evolved to a full-fledged research project that seeks to formalise the interface between software and hardware to ultimately generate provably correct code that handles the delicate interaction with today's complex hardware.

The combination of the idea behind Sockeye and the insights into BMC management provided by the Enzian platform have given rise to the model-mechanism based approach to power and clock management pursued in this thesis.

Chapter 3

Power and Clock Management

In this chapter, we take a closer look at how power and clock management is conducted in a modern computer system, setting the scene for the work and results of this thesis. Over the course if this chapter, we flesh out several observations, assumptions and definitions about power and clock management that we will refer to when motivating design choices in our later work.

Literature on power and clock management from a general perspective is very scarce. As already mentioned in the introduction, most of the following discussion is hence based on observations made in connection with the Enzian platform.

This chapter is structured as follows: In the first section, we define clock and power resources and develop a more concrete perspective on the concept of *power and clock management*. In subsequent sections, we approach the topic from a functional point of view. We identify and elaborate on the entities in a computer system that adopt the following management roles (see figure 3.1 for a schematic overview):

- **producers** that supply specific resources (section 3.2)
- **consumers** that require specific resources (section 3.3)
- **managers** that are balancing the demands of the consumers and the supply of the producers (section 3.4)

In order to simplify the reasoning about all entities involved in power and clock management, we furthermore define the term *platform*, inspired by [10], and *component* in the context of this thesis as follows:

Definition 1 (Platform)**.** A platform is the composition of all producers, consumers and managers present in a computer system.

Definition 2 (Component)**.** A component is a single producer or a consumer.

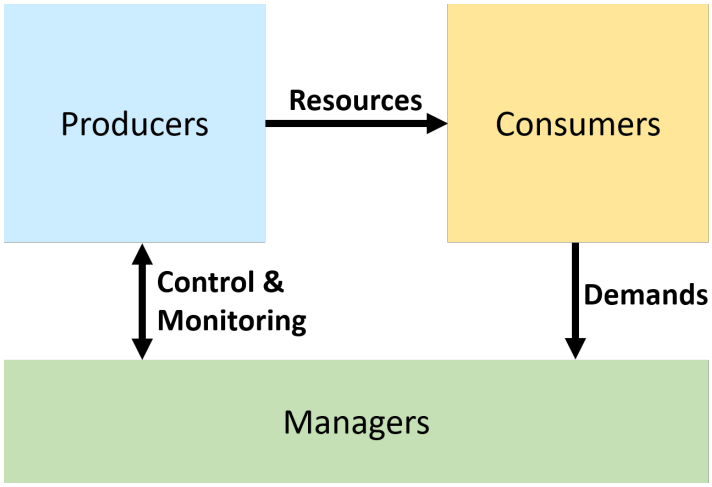


Figure 3.1: A schematic overview of the relation between producers, consumers and managers

The Component definition purposely excludes managers, since the platform model we are going to develop over the course of the next three chapters adopts a manager's perspective on the platform.

3.1 Resources

In the context of this thesis, a resource is defined as follows:

Definition 3 (Resource)**.** A resource is a clock signal, a logical signal or a power source.

Remark. From a technical point of view there is no difference between a conductor supplying power and one that is transmitting a logical signal. In this thesis, we are going to use the term *logical signal* in situations where only the interpretation of the power signal as being high or low is of import.

Clock signals and power are transmitted over some conductor, in computer systems this is most commonly a wire or a circuit trace.

Remark. In the context of this thesis, we refer to *conductor* strictly in a power and clock management context. When referring to a conductor, we thus always mean a conductor that is supplying a clock signal or power to the platform.

From an alternate perspective, we could consider a conductor to embody the corresponding resource. Power and clock management hence translates to *managing the state of conductors*.

Before we can define Power and Clock Management for our purposes, we need to think about our performance metric. Management objectives are usually tied to some performance metric that is used to compare and evaluate different management strategies.

Power and clock resources are critical in the sense that providing a component with a wrong input can be devastating. For this reason, components usually define the following electrical specifications:

- absolute maximum ratings: constraints that must be observed at all times to prevent the component from taking permanent damage
- recommended operating conditions: stricter constraints that must be observed to ensure the correct operation of a component.

Based on these specifications, we define the term *input constraints* as follows:

Definition 4 (Input Constraints)**.** At any given point in time, the input constraints of a component are defined by its *recommended operating conditions* if the component is supposed to be operational at this point and its *absolute maximum ratings* otherwise.

Remark. We refer to these constraints as *input* constraints since in all generality, a component can exert no direct control over its inputs, while it can very well influence the character of its outputs. For this reason, both absolute maximum ratings and recommended operating conditions are mainly concerned with the component's inputs.

For the reasons mentioned above, we argue that in a power and clock management scenario, meeting all input constraints takes priority over the usual efficiency and utilisation criteria.

In the remainder of this thesis, we hence will adopt this more concrete view on power and clock management:

Definition 5 (Power and Clock Management)**.** Power and Clock Management is concerned with managing the *state of conductors*, in a fashion that respects every component's input constraints while satisfying consumer demands.

Remark. On some platforms, there might exist certain consumer demands that cannot be fulfilled without violating any input constraints. In the context of this thesis, we assume that power and clock management is allowed to *fail* by refusing to comply in the face of such demands, rather than striving to provide a good approximation.

The state of conductors is given by their measurable electrical characteristics such as voltage, frequency and current. Commonly, voltage and frequency can be regulated to a stable, static value by the platform's producers. The current that is flowing across a conductor at any given point in time, however, depends on the dynamic power requirements of the attached components. These are in turn determined by the load the system is experiencing.

Because of this difference, we refer to voltage and frequency as *stable* conductor characteristics whereas current is *volatile*.

Using these concepts, we can further classify power and clock management into two different categories:

Definition 6 (Static Power and Clock Management)**.** Static Power and Clock Management is about managing the *stable* characteristics of conductors.

Static Power and clock management can be accomplished directly by appropriately configuring the outputs of the platform's producers.

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.

This is usually accomplished as follows: the overall status of the platform is continuously monitored. Whenever a fault or alert condition is detected, appropriate countermeasures are taken. If, for instance, the current across a conductor exceeds a certain threshold, power-limiting measures such as CPU throttling or an emergency shutdown are initiated.

3.2 Producers

From a physical point of view, the term *transformer* might be more appropriate than *producer*: Naturally, any entity providing power or clock signals also consumes power in one form or another. In our management context, we use the concept of a *producer* with respect to an entity's main purpose in the context of the platform and hence define it as follows:

Definition 8 (Producer)**.** A producer is an entity whose main purpose is the production of platform resources, i.e. specific states on power or signal-carrying conductors.

In a modern computer system, producers are electronic circuits such as power supplies, voltage regulators, oscillators and clock generators.

As described in [16], clock producers usually form a so-called Clock Tree. Similarly, for several reasons that are detailed very concisely in ([10], p. 121), producers concerned with power resources are commonly arranged in a multi-level hierarchy. This results in the following observation:

Observation 9. Producers are arranged in hierarchies.

We conclude this section with some more observations we have made when surveying the producers present on the Enzian's platform.

A producer is usually a distinct unit that is soldered onto the circuit board. The structure of a producer is defined by the fixed layout of its pins, which are used to integrate the producer into the platform by connecting the appropriate conductors. The purpose of a pin and if it serves as an input or output is specified in the producer's manual.

Observation 10. Producers feature a fixed set of pins that can be connected to conductors. Each pin serves a pre-defined purpose.

Generally, producers are focused on providing *stable* states on the conductors connected to their output pins. They normally do so using a feedback loop that takes corrective actions if the true value deviates too much from the required one.

Observation 11. When directed to supply a certain value of a stable conductor characteristic, a producer will continue to output this particular value until it is told otherwise, unless a fault condition occurs.

Having multiple, independent producer regulate the state of the same conductor does not make a lot of sense; imagine two producers attempting to regulate the voltage across said conductor to different values. An exception to this is the case where all such producers can only be regulated jointly; for instance the up to four slaves operated by a MAX20751. However, in such a situation we would also be modelling all such producers as a single, joint producer. Thus we arrive at the following observation:

Observation 12. Any platform conductor's state is generated by at most one producer. Consequently, any conductor is an output to at most one producer.

The next conductor state a producer is required to provide is usually difficult to anticipate; a user might press the power or sleep button at an arbitrary point in time and a sudden increase in a CPU's workload might prompt it to request an increase in its clock speed. For this reason, a producer should be able to provide the whole range of output states it can offer at any given point in time, regardless of the previous input and command history.

Observation 13. In most situations, the range of output states a producer can provide is only dependent on its current inputs and *not* on its previous input and command history.

Remark. There are situations where this is not the case, most notably in connection with bus communication. See section 5.6 for a detailed discussion of this.

3.3 Consumers

Definition 14 (Consumer)**.** A consumer is an entity of the computer system that needs to be supplied with specific power and clock inputs but is itself not mainly concerned with the production of platform resources.

Remark. Contrasting this definition with said of a producer (Definition 8) it is clear that the two are deliberately mutually exclusive.

The granularity of what is considered to be a single consumer is a matter of perspective. On the Enzian, we could for instance identify the ThunderX CPU, its DRAM banks and I/O ports as one consumer entity. In terms of complexity, consumers are commonly several levels above comparatively simplistic producers and require much more specific power and clock inputs.

We conclude this section by making a few more observations about the nature of consumers. Again, as in the producer section, these are based on the Enzian platform.

With the introduction of dedicated hardware for power and clock management, such as BMCs, consumers can be designed in a platform-independent manner. It follows that:

Observation 15. Apart from interfaces to managers and the state of conductors serving as inputs, the platform is generally transparent to consumers.

In order to receive the correct inputs, consumers thus impose specific demands on them. (See section 3.4.1 for a discussion on how said demands might be communicated to the platform's managers.) Some such demands are the result of different power states a consumer can adopt:

Observation 16. Consumers usually define a set of power states and transitions between them. Each such power state and each transition step is associated with very specific power and clock inputs that the consumer requires in that particular situation.

For example, the ThunderX's manual defines a 7-step power-up sequence that has it transition from a *powered-down* to a *powered-on* state ([7], p. 1827).

Furthermore, inputs to powered-on consumers can usually be fine-tuned to achieve different efficiency and performance trade-offs; an example for this is frequency and voltage scaling. Hence:

Observation 17. A consumer might also dynamically demand changes to its inputs, for instance in response to changes of its workload.

Remark. Although it would be more precise to say that the *software* running on a consumer requests such dynamic changes, this distinction is not required for our purposes.

We have yet to discuss the meaning of consumer demands from a time perspective. Conceptually, a consumer demand is fulfilled if the state of the consumer's input conductors agrees with the demand. In the context of power and clock management, it is quite obvious that only *transiently* adhering to a demand is insufficient; a single voltage pulse will clearly not be able to adequately power a CPU. It hence follows that a consumer demand needs to be imposed on the corresponding input conductor state permanently, until it is replaced by a more recent demand.

Observation 18. Consumer demands remain in effect until replaced by more recent demands.

Remark. This nicely coincides with observation 11.

3.4 Managers

This section discusses the manager role of the platform, including the instruments and mechanism a manager commonly has at its disposal to fulfil its purpose.

Definition 19 (Manager)**.** A manager is an entity concerned with the provisioning of correct conductor states. It is aware of the consumer demands and controls and configures the producers accordingly.

As stated in the introduction, the BMC is commonly heavily involved in this manager functionality. Since we wish to explore power and clock management strictly in the context of the BMC, we need to make the two assumptions about our platform.

Assumption 20. There is exactly one manager on the platform.

Nothing prevents the aggregation of multiple managers in a single system. However, letting the same resource be managed conjointly by several managers will most likely result in redundancy and inefficiency, so by stretching the definition of *platform* a bit, we could most likely identify different, possibly hierarchical management domains that we could consider to be individual platforms. For the purposes of this thesis, however, we will think of our computer system to be a *regular* server system featuring a single manager entity.

Assumption 21. This manager is a BMC.

That this is the case is not set in stone either: the ThunderX for instance, was originally intended to provide a *Voltage Regulator Module* that would assume direct control of the voltage regulator supplying the CPU's core voltage ([7], chapter 40).

This has another implication for the BMC: In general, it cannot manage its own power and clock inputs, since said are commonly a pre-condition for the BMC's management firmware to be able to run correctly. It hence follows that:

Observation 22. The BMC is not a consumer of the platform it is managing.

Remark. As mentioned in the introduction, the BMC is itself a microcontroller of non-negligible complexity. This of course raises the question: who is managing the BMC's power and clock inputs? This might be the task of another, simpler management domain from the perspective of which the BMC is a regular consumer. Or, alternatively, such inputs might be provided in an *always on if the computer system is plugged in* fashion. In any case, the bootstrapping and input management of the BMC is out of the scope of this thesis.

Making use of the above assumptions, we can now discuss the management mechanisms and instruments of a BMC specifically. We split this discussion into two parts, one concerned with consumer awareness and the other with how the BMC exerts control over the producers.

3.4.1 Consumer Awareness

To correctly manage the platform, a BMC must be aware of the consumer's power and clock demands. This happens either in an active fashion, whereby the consumer actively communicates with the BMC or in a passive manner using hard-coded information.

Consumer - BMC communication

There is a variety of different interfaces that seek to enable consumer-BMC communication. The most prevalent such interfaces include the Advanced Configuration and Power Interface (ACPI) [8] and the Platform Environment Control Interface (PECI) ([10], p. 97). Also notable is the Intelligent Platform Management Interface (IPMI) [14], although said aims to provide remote management capabilities to system administrators rather than enabling consumer-BMC communication.

Hard-coded information

By definition, consumer requirements cannot always be obtained in an active manner: an integral part of power management is the correct bootstrapping of consumers, during which they commonly are not able to communicate their power requirements. Therefore, the bootstrapping requirements are defined in the corresponding consumer's manual and are typically hard-coded into the BMC's management firmware.

3.4.2 Producer control

Successful power and clock management requires two-way communication between the BMC and the producers: In the context of static management, the BMC must be able to transmit commands to correctly configure the producers. In order to perform dynamic management, it must be able to receive feedback from them to continually monitor the platform activity and react to potential failures.

How exactly this communication is realised is subject to the platform's design and the nature of the producers.

Enzian relies primarily on the general purpose input/output (GPIO) pins exposed by the BMC. These pins are configurable by the software running on the BMC and can be used as detailed in the following subsections.

GPIO as output

GPIO pins can serve as both logical inputs and outputs. If a pin is configured to be used as an output, the management software can set the pin's logical level to either high or low. In such a case, we could argue that the BMC itself also adopts the role of a producer, since the production of such a logical signal constitutes an integral part of its *management* purpose. We thus arrive at the following observation:

Observation 23. The BMC is a producer

GPIO as input

A logical input to the BMC can by itself only provide a very limited amount of information (high or low) and is hence commonly used as alert or fault signal, to indicate

an (imminent) producer failure. As part of dynamic power and clock management, the BMC will then appropriately react in an event-based manner.

GPIO and I²C bus lines

GPIO pins can also be connected to I²C bus communication lines. I²C is a bus communication protocol that allows communication between multiple devices connected to the bus. Quite a few producers are I²C compliant, most by implementing the higher level System Management Bus (SMBus) or Power Management Bus (PMBus) protocols that build on I²C and SMBus respectively.

Every such bus-compliant device defines a set of commands that the BMC can issue, both to query the current status of the device, including the readings of any integrated sensors, and to control its power or clock outputs. The I²C bus is thus an instrument that can be used in both a dynamic as well as a static management context.

The situation is, however, not quite as simple as it seems at first glance. The reason for this is I²C's solution to the following problem: If several devices concurrently try to pull the line up to transmit a 1 and pull it to ground to transmit a 0, this may cause a short circuit.

I²C avoids such a scenario as follows: Every bus line is pulled up to a supply voltage level using a pull-up resistor. In order to communicate, devices may only pull the line low. [11] At the same time, the I²C specification does not prevent powered-off devices from pulling the bus lines low, which effectively renders the bus unusable until all such devices are powered.

Observation 24. The I²C bus is not necessarily operational if any connected devices are powered off.

3.5 Summary

In this chapter, we have discussed platform-level power and clock management. We have realised that there is a direct correspondence between the resources managed in this context and the state of platform conductors. We have then identified two types of physical characteristics that comprise the state of such conductors: stable characteristics that can be regulated to fixed values by the platform, and volatile characteristics that change dynamically. Based on this distinction, we have subcategorised power and clock management into static and dynamic management.

Additionally, we have defined three relevant roles in this management context: Producers, Consumers and Managers. Using the Enzian platform as an example, we have made some observations about the general nature and behaviour of entities adopting these roles. We have discussed the means platform managers have at their disposal to monitor and communicate with the other components. With the observation that I²C buses are not

necessarily operational, we have encountered a first subtle issue that our management solution must address.

Chapter 4

Definition of Scope and Objective

In this chapter, we concretise the scope of operation and the objective of the thesis.

4.1 Scope

In this thesis, we are only going to address the topic of *Static Power and Clock Management*, as it is defined in section 3.1. The reason for this is that, assuming ideal conditions and every component working as expected, static management is sufficient to correctly operate the platform. However, it goes without saying that in a *real* setting, the correct handling of potentially catastrophic events as performed by dynamic management is extremely important for platform reliability and safety and must not be omitted.

As already hinted at, for static management to be reasonable by itself, we need to make the following assumption.

Assumption 25. The platform is operating as expected and no exceptional events are occurring.

Thus, neither input constraints nor consumer demands involving *volatile* characteristics are of any concern to us. In order to simplify our reasoning about static management, we redefine the *state* of a conductor to only refer to its *stable* characteristics.

Based on the definition of stable characteristics, as well as our assumptions about the BMC being the platform's only manager, we can furthermore make the following assumption:

Assumption 26. Once the BMC stops altering the configuration of producers, the state of the platform's conductors will eventually stabilise and will remain unchanged until the BMC issues further changes.

If all of the platform's conductors feature such a stabilised state, we will call the collection of all such states a *stable platform state*. For the purposes of this thesis, we are in need of two more assumptions:

Assumption 27. Consumer demands are time-invariant.

It is not unusual for power sequences to feature timing constraints. The power-up sequence of the ThunderX, for instance, requires that a signal indicating *clock readiness* is asserted at least 3 milliseconds after the corresponding clock input has stabilized at its target frequency [7].

For our purposes, however, we are going to incorporate such timing requirements in the scheduling of consumer demands rather than the demands themselves. In the case of the ThunderX, we will only begin to fulfil the demand "Assert clock readiness" when at least 3ms have passed since we have reached a stable platform state after fulfilling the demand that had the clock input transition to the mentioned target frequency.

Assumption 28. Consumer demands are not associated with any real-time constraints. In particular, the BMC is allowed to wait until the platform has reached a stable state before issuing further changes.

Arguably, most timing-sensitive tasks belong to the dynamic power and clock management domain, which renders this assumption not unrealistic. Furthermore, considering the platform-independent design of consumers (as indicated by observation 15), adding very harsh real-time constraints to power sequences will severely limit the range of platform designs that will be able to uphold said, and will thus most likely be avoided if possible. Indeed, on the Enzian platform neither the ThunderX's nor the FPGA's power sequences feature any real-time constraints.

In spite of this assumption, we consider a static management strategy that stalls the realisation of consumer demands indefinitely as incorrect.

In the context of these assumptions, we can further concretise static management: With consumer demands being *permanent* (observation 18) and time-invariant and in the absence of real-time constraints, static management will only be required to take action *upon a change in consumer demands*:

Definition 29 (Static Management)**.** Upon a change in consumer demands, static management must construct a sequence of actions to be executed by the BMC such that:

- the platform transitions to a new stable state that agrees with the new consumer demands
- no input constraints are violated

No further static management actions must be performed until the consumer demands change once more.

4.2 The Objective

As mentioned in the introduction, we wish to explore an approach to a subset of Power and Clock Management that separates policy from mechanism. Now that we have precisely defined said subset in the last section, we can reason about the precise nature of the already mentioned model and mechanisms that we aim to employ for this purpose.

The information our model reflects must be sufficient to support our intended mechanism. In the following two subsections, we are going to first discuss the nature of the mechanism we require before taking a closer look at the model.

4.2.1 The Mechanism

Pooling all the information from our definition of scope (4.1) and chapter 3, we can come to more conclusions about the function and nature of our mechanism:

First of all, we can naturally split the objective of static management as defined by definition 29 into two sub-goals, which we will discuss in more detail in the following two subsections:

1. **State Generation**: The process of determining a *new stable platform state* that observes the new consumer demands without violating any input constraints.

2. **Sequence Generation**: Generating the sequence of BMC actions that will have the platform transition from the current stable state to this *new stable platform state*.

State Generation

To see why this is necessary, we need to remind ourselves of observation 9, which stated that producers are generally arranged in hierarchies. Consumers are mainly supplied by the producers in the lowest level of these hierarchies. The state of the conductors in the upper part of the hierarchy is hence generally transparent to consumers and thus not explicitly constrained by consumer demands. Since the interconnections between different hierarchy levels can be almost arbitrarily complex and convoluted, assigning an appropriate state to each conductor is not necessarily a trivial problem. This is further amplified by the fact that the platform's input constraints (see definition 4) are dependent on the required conductor states.

Sequence Generation

Not all of the BMC's actions can be performed in an arbitrary sequence. A good example are commands that are to be transmitted using the I^2C bus. Before such a command is sent, we must make sure of the following: The intended receiver should be powered, since powered-off devices are usually not responsive to bus commands. Furthermore, as detailed by observation 24, we must ensure that the I^2C bus is operational. Consequently, we must be able to order the BMC's actions in a manner that observes all such restrictions.

Consumer Demand Generation

There is another problem our mechanisms must solve. In the definition of static management, we implicitly assume that we are aware of the *new consumer demands*. When we are required to have multiple consumers transition to another power state simultaneously, this is not necessarily the case: Not every platform is built in a manner that allows different consumers to transition to another power state independently of the others. Therefore, we also add *Consumer Demand Generation* to our required mechanisms:

> **Consumer Demand Generation**: Generate a feasible interleaving of all consumer transitions that are requested simultaneously.

4.2.2 The Model

Our model should be able to capture how a platform behaves. It is, however, not supposed to describe the management behaviour of the BMC itself, since it is precisely this behaviour we wish to correctly recreate using our model and mechanisms. We are thus required to devise a model that captures the platform's behaviour from the management perspective of the BMC.

4.3 Summary

In this chapter, we have narrowed down the scope of this thesis to static power and clock management. Under the assumption that every platform component is operating as expected, we have concretised our definition of static management. Based on this definition, we have identified three mechanisms necessary to generate correct static management actions based on a suitable platform model.

Chapter 5

Modelling the platform

This chapter is concerned with the nature of the platform model we have devised to achieve the objective defined in the previous chapter.

5.1 Description of Structures

Before we define our platform model, we wish to introduce some notation in order to unify the descriptions of certain structures found in our model. Since we aim to build mechanisms directly on top of our model, not only is the semantic meaning of the different concepts important, but also the limitations imposed by the representation of the information, i.e. the syntactical aspects. Consequently, we are not merely describing a platform model but rather a modelling language for a platform.

In this chapter, we will represent structures in the following manner: Let S be some structure, composed of attributes a_1 to a_n. We will provide a schematic overview of the structure in a form as displayed by table 5.1.

As in object oriented programming, we will use the notation S.a_1 to indicate that we access attribute a_1 of a structure S. To increase the readability of subsequent code and formulae, attribute names are kept short. We thus rely on the *designation* column to provide a meaningful and human-readable identifier for the defined attribute.
Occasionally, we will encounter nested structures. We denote the *syntactical structure* of any substructure with its italicised name.

Let *Structure* be some syntactical structure. We use the notation {*Structure*} to denote a set of structures *Structure* and [*Structure*] to denote an ordered list.

Structure S		
a_1	designation for a_1	*syntactical structure of* a_1
	...	
a_n	designation for a_n	*syntactical structure of* a_n

Table 5.1: Schematic representation of structures

A schematic overview of an instance I of our structure S would have the following format, whereby terms in <> serve as descriptive placeholders:

I [S]	
a_1	<instance of a_1 >
...	
a_n	<instance of a_n >

Table 5.2: Schematic representation of structure instances

5.2 General structure

For the perspective of the BMC, the platform is an *ensemble of components* with associated management actions. This natural structure is retained by our model: we describe a platform's behaviour by describing the behaviour of the involved producers (section 5.4) and consumers (section 5.7), along with information on their composition (section 5.8).

This separation of descriptions is possible because every component has but a local view on the platform: both producers as well as consumers (see observation 15) are only aware of the state of conductors that are directly connected to them. Thus, their behaviour can be described in isolation, independently of the whole platform composition.

This model structure has two main advantages: Firstly, a component's behaviour in isolation is easier to understand and is usually reasonably well-documented in its manual. Secondly, we are only required to describe each type of component once, to then be able to build arbitrary platform instances with it.

As an example, consider the platform instance depicted in figure 5.1: It features five producer instances A to E and two consumer instances CPU and $FPGA$. Recalling that we denote the *structure* a particular instance is based on with square brackets [], as defined section 5.1, it follows that instances A and B as well as D and E have been generated by the same producer description $P1$ and $P3$ respectively.

5.3 Representation of conductor state

This sections aims to discuss the representation of conductor states we use in our platform model.

As detailed in section 3.1, the state of a conductor is given by the values of its physical characteristics. This definition provides a quite natural structure for this section:

In a first subsection, we will think about how to represent a single such characteristic. In a second subsection, we will discuss how to combine representations of several such characteristics to represent the complete state of a conductor.

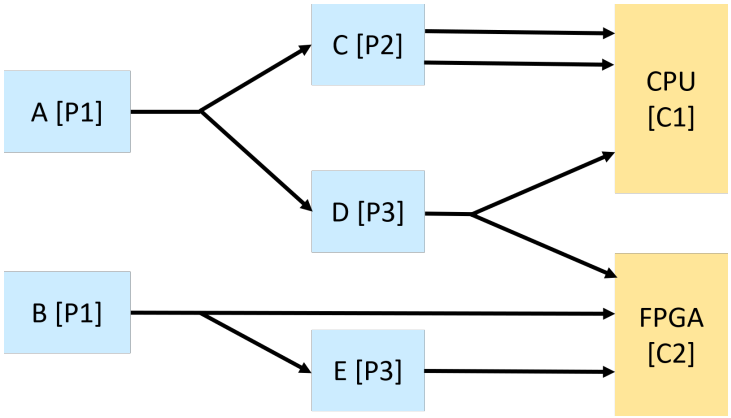


Figure 5.1: A schematic example of a platform instance, with producer and consumer instances drawn as boxes and the composition information abstracted as black arrows.

5.3.1 Representation of a characteristic

As is the case with most physical quantities, characteristics of conductors are of an inherently continuous nature. However, measurements and regulations performed on them are always of limited accuracy. For this reason, as well as to avoid precision and efficiency concerns associated with real-valued representations, we apply a discretisation to each such characteristic (see table 5.3).

For instance, the typical precision of voltage regulators found on platforms is in the millivolt range. Thus, the voltages across conductors are described by a natural numbers representing *millivolts*. Similarly, the crystals used in oscillator circuits commonly produce a "sine-wave typically ranging from 32kHz to 50MHz". [16]

characteristic	representation
voltage	$\mathbb{Z}$ (interpreted as millivolt)
frequency	$\mathbb{N}$ (interpreted as kilohertz)

Table 5.3: The discretisation of stable characteristics

5.3.2 Representation of state

There are two reasons why the combination of characteristics to the state of a conductor warrants special consideration:

On one hand, as discussed in section 4.2.2, our model must be capable of representing all *possible* conductor states. Because of this, we need to describe *state spaces* rather than individual states. For the purpose of this model, we will make use of the most general (and obvious) way to describe such a state space: we simply specify the whole set of individual conductor states present in the given state space. The actual implementation

introduces some abbreviations to improve the conciseness of state space descriptions, see section 8.1.1.

On the other hand, the characteristics of interest to us depend on the purpose of the conductor. As detailed in section 4.1, we are only concerned with *stable* characteristics, which happen to be voltage and frequency. Most generally, the state of a conductor can be described by the value of its low voltage characteristic, its high voltage characteristic and the value of the frequency characteristic with which it is alternating between:

Conductor State		
low	low voltage characteristic	$\in \mathbb{Z}$
high	high voltage characteristic	$\in \mathbb{Z}$
freq	frequency characteristic	$\in \mathbb{N}$

Table 5.4: The Conductor State structure

Remark 30. Of course the shape of the alternation is crucial too, if it is a sine function as with alternating current or clock-shaped; for our purposes, we will simply push this problem to the implementation.

As detailed above, we define the type of a conductor state space as follows:

$$\textit{State Space} \subseteq \{\textit{Conductor State}\}$$

We will encounter *State Space* as a description for syntactical structures (see table 5.1) several times in subsequent sections.

However, the State Space S of a conductor W transmitting direct current can be described more concisely: Let E be an element of S. Then E.freq should be 0 Hz and E.low = E.high. Therefore, a single voltage characteristic would be sufficient to describe E. Similarly, the State Space of a conductor transmitting a logical signal could simply be described as a subset of {low (0), high (1)}. To prevent our modelling examples from becoming too unwieldy, we will therefore represent a State Space S as follows:

$$\mathrm{S} \subseteq \begin{cases} \mathbb{Z} & \text{if W carries direct current} \\ \{0,1\} & \text{if W is transmitting a logical signal} \\ \{\textit{Conductor State}\} & \text{otherwise} \end{cases}$$

5.4 Producer Description

In this section, we take a closer look at how we can describe the behaviour of producers as defined in section 3.2. As already established in said section, producers feature a fixed layout of pins, serving as connections for pre-defined inputs and outputs to the producer.

Because of this, it makes sense to describe a producer's behaviour in an *input/output-centred* fashion.

As we have established with observation 13, the stable states a producer can apply to its output conductors are mostly independent of the history of inputs and commands. For the purposes of this model, this means that we are generally not required to maintain any additional internal producer state. Thus, the descriptions of the behaviour of input and output pins are sufficient to model the behaviour of the entire producer:

Consider therefore a producer with n input pins named IN1, IN2, .., INn and m output pins named OUT1, OUT2, .., OUTm. Our producer description will then be of the following format:

Producer		
IN1	description of IN1	*Input*
...		
INn	description of INn	*Input*
OUT1	description of OUT1	*Output*
...		
OUTm	description of OUTm	*Output*

Table 5.5: The Producer structure

Remark. We choose our attribute names to correspond to pin names for reasons of simplicity: In future sections, we will need to be able to retrieve the pin description that corresponds to a given pin name.

As their names indicate, the Input and Output structures referenced in the producer schematic describe individual input and output pins. We will discuss these structures in the next two sections. Unsurprisingly, their focus is on the *states* that connected conductors can or are allowed to attain.

In order to prevent this section from becoming too abstract, we are going to develop the description of an example producer alongside the abstract producer structure. This example producer is the MAX15301 [13], a voltage regulator produced by Maxim Integrated. It produces a single direct current output featuring a voltage that can be specified via PMBus.

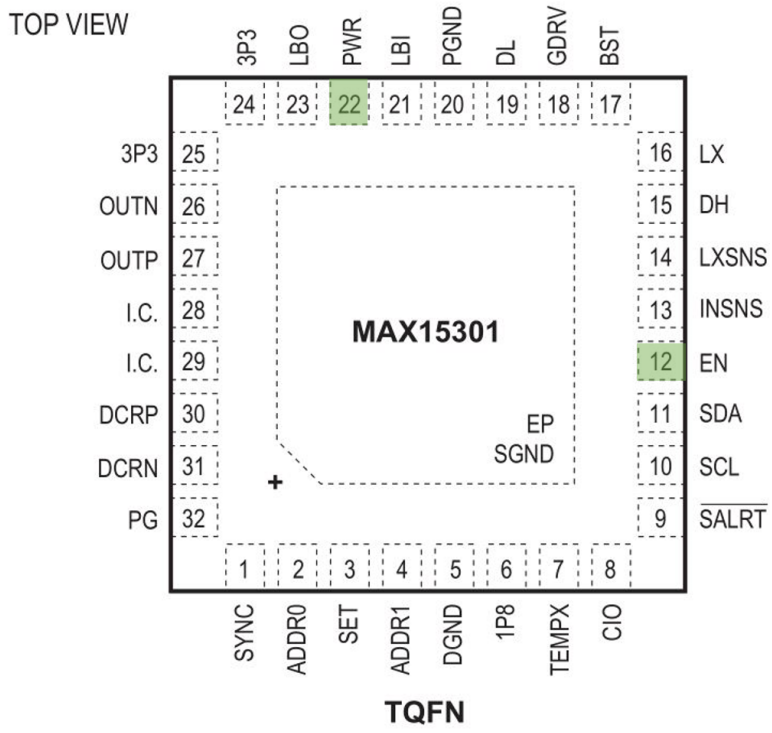


Figure 5.2: pin layout of the MAX15301, taken from its manual ([13])

Looking at the 32 pins the MAX15301 defines (see figure 5.2), the prospect of devising a description of it seems daunting. Luckily, almost all of these pins are irrelevant for our purposes: some just need to be connected to ground in a specific manner to ensure correct operation, others offer points of connection for additional sensors and so on. For us, the important pins (coloured in green in figure 5.2) are:

- **PWR**: an input pin that serves as power supply to the regulator.
- **EN**: an input pin that serves as a so-called *enable* signal: if the voltage across it is interpreted as HIGH, the MAX15301 will regulate the output voltage to a desired value, and if the voltage is interpreted as LOW it will not.

As an attentive reader might have noticed, we have not identified the output pin whose voltage the MAX15301 is supposed to regulate. The reason for this is that our vision of a producer as a stand-alone integrated circuit starts to crumble a bit here: In a section called *Design Procedure*, the MAX15301's manual meticulously describes how we must combine several of its inputs and outputs with additional capacitors and MOSFETS and whatnot to obtain **the** output the MAX15301 is capable of regulating. The electrical properties of this additional circuitry are far too complex to toss at our platform model (let alone a computer science bachelor student, such as myself), so we take the only reasonable course of action:

We pretend that all this circuitry is a fixed part of the MAX15301 and define an additional, *virtual* output pin **OUT** that provides **the** output.

As far as discussed, the producer instance corresponding to the MAX15301 look as follows:

MAX15301 [Producer]	
PWR	< Input instance >
EN	< Input instance >
OUT	< Output instance >

5.4.1 Producer Inputs

This subsection is concerned with the description of individual input pins to the producer. There are several pieces of information that we must include in such a description:

Input		
amr	absolute maximum rating	*State Space*
mon	monitor functionality	*Monitor Action*

Table 5.6: The Input structure

As detailed in section 3.1, producers define an **absolute maximum rating** for each of their inputs, which must be observed at all times, independently of the states attained by other inputs or outputs of the producer.

Occasionally, a producer offers **monitor functionalities** on some of its connected inputs. These functionalities are especially important in the context of the sequence requirements: In order to correctly time producer commands, our management solution must be able to query the state of conductors to determine if previously issued commands have already taken effect. In section 5.5, we will discuss the usage of these functionalities in more detail.

Now that we have defined the nature of Input structures, we can add the descriptions of the MAX15301's two input pins, EN and PWR.

We wish to remark on the consequences of modelling a conductor as a logical signal, using the example of the MAX15301's EN input pin: as already mentioned in the description of this pin, only the interpretation of the voltage across the EN input as HIGH or LOW is of import. It is likely such an EN pin is connected to a GPIO pin of the BMC (see section 3.4.2), which can only be set to either HIGH or LOW. If this were the case, modelling the EN input as a logical signal would be the ideal choice: we convey exactly the right amount of information and need not be bothered with exact voltage values. The drawbacks of this choice are just as obvious: if the EN input is connected to an output whose voltage value needs to be set by the BMC, the EN input description must include information on the interpretation of different voltages and thus cannot be modelled in a logical fashion.

For now, since we consider the MAX15301 in isolation, nothing speaks against modelling it as a logical input. We thus define its absolute maximum rating as the set $\{0, 1\}$ in

accordance with the State Space abbreviations defined at the end of section 5.3. Since no monitoring functionality on the state of EN is defined, we leave the **mon** field empty.

The maximum absolute rating of the PWR input is defined to be -0.3V to 18V. The PMBus command "READ_VIN" returns the voltage across the PWR input.

Entering this new information into our schematic MAX15301 instance yields the following:

MAX15301 [Producer]		
PWR	amr	$\{-300, \ldots, 18000\}$
	mon	READ_VIN
EN	amr	$\{0, 1\}$
	mon	
OUT	< Output instance >	

5.4.2 Outputs

When designing our platform model, we have decided to represent most of a producer's behaviour in the Output structures it defines. The main motivation behind this decision is observation 12: while a conductor might serve as input to several components, it will be connected to at most one output pin of a producer. It thus makes sense to pool as much information as possible at the output pins, rather than composing it from various input pins later on.

In light of this, the distinction between the output pin and the conductor connected to it is becoming somewhat pedantic. In our descriptions, we will therefore occasionally be referring to the unique conductor connected to an output pin as the *output conductor*.

The schematic overview of the Output structure is defined as follows:

Output		
set	state set method	*Conductor State* $\rightarrow$ *Command Action*
poss	state possibilities	$\{$*State Possibility*$\}$

Table 5.7: The Output structure

The **state set method** defines a function that, given a desired stable state S of the output conductor, returns an action the BMC can perform to have the conductor transition to state S.

As the syntactic structure indicates, the **state possibilities** are a set of State Possibility structures. These structures form the heart of the producer descriptions: they identify a state space and a collection of requirements. The semantic meaning is as follows: every state in the described state space *can* be attained by the output conductor, provided that all given requirements are fulfilled.

State Possibility		
state	output state space	*State Space*
seq	sequence requirements	*Initial State* $\rightarrow$ *Event Graph*
req	state requirements	{*State Requirement*}

Table 5.8: The State Possibility structure

The **output state space** identifies that aforementioned state space, which can be adopted by the output with which this State Possibility is associated.

The **sequence requirements** describe how the BMC can cause the state of the output conductor to transition to a state in the output state space. We will discuss the nature of this description in section 5.5.

As the term indicates, the **state requirements** are a collection of requirements on the state of other conductors. The State Requirement structure is defined as given by table 5.9. The amount of nesting of structures might seem excessive at this point; however, in later sections we will profit immensely from clearly defined attributes.

State Requirement		
id	a pin identifier	*string*
state	a state space	*State Space*

Table 5.9: The State Requirement structure

Because of the *local view* each component has on the platform (see section 5.2), the only other conductors that may be mentioned in **req** are conductors connected to one of the pins defined by the producer. Recalling the pin names of our producer we defined in the beginning of this section, the following must evaluate to true:

$$\forall r \in \text{req}.\ r.\text{id} \in \{\text{IN1}, \ldots, \text{INn}, \text{OUT1}, \ldots \text{OUTm}\}$$

In terms of static management, the purpose of State Requirements is the representation of *Input Constraints* as specified by definition 4 in section 3.1. However, as discussed above, we also allow other output pins of the consumer to be referenced.

Closely studying the State Possibility structure reveals that all of the declarations of State Spaces it allows are *absolute*: none may exhibit any dependence on the initial platform state or any other kind of condition. This is in accordance with observation 13, which stated that the stable states a producer can apply to its output conductors are generally independent of the history of inputs and commands.
With all these concepts in place, we can return to our example producer: We assume that our platform's electrical ground is at 0V. Furthermore, since we have established that every conductor is only driven by at most one producer (see observation 12), we conclude that if the MAX15301 is not regulating the voltage across its OUT output, said should drop to 0V. From the MAX15301's manual, we can extract the relation between

its inputs EN and PWR and its output OUT, which we have visualised in the following level plot:

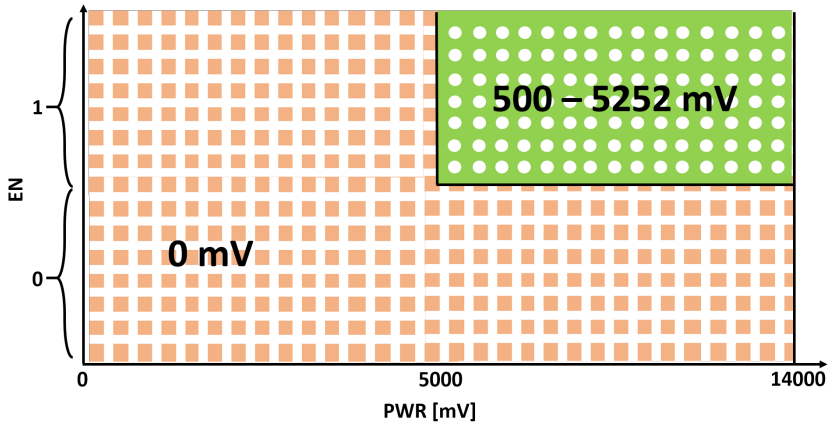


Figure 5.3: The relation between EN, PWR and OUT as a level plot

The State Possibilities we describe must be able to capture the two levels visible in the plot. We could for instance identify the following four regions P1 to P4 (see figure 5.4) and describe each as an individual State Possibility.

Note that it is not possible to correctly describe the red (0V) level using only one State Possibility. The reason for this is that State Requirements only allow us to place **independent** restrictions on different conductors, which means that we can only specify (potentially multiple, but then disjoint) *convex* regions. We could, however, do with only three State Possibilities, but for reasons of symmetry we will stick to four.

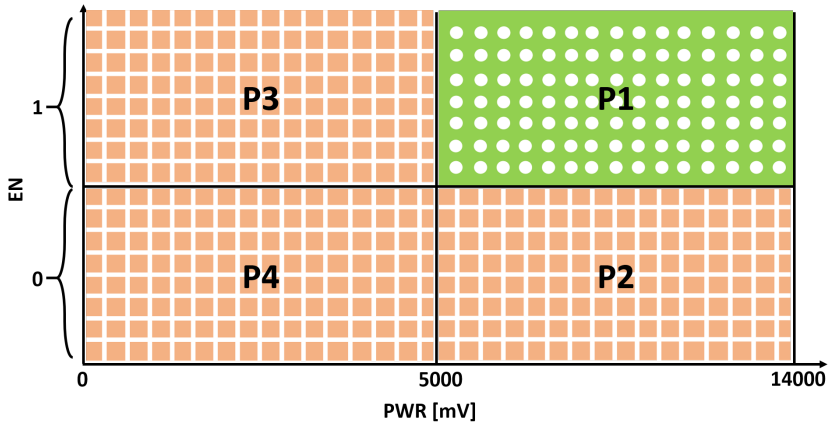


Figure 5.4: Regions which we will describe as individual State Possibilities.

Filling in the descriptions of P1 to P4, we obtain the schematic instance as depicted by table 5.10.

MAX15301 [Producer]					
PWR	amr	$\{0, \ldots, 14000\}$			
	mon	VIN_STATUS			
EN	amr	$\{0, 1\}$			
	mon	None			
OUT	poss	state	$\{500, 5252\}$		$P1$
		req	id	PWR	
			state	$\{5000, \ldots, 12000\}$	
			id	EN	
			state	$\{1\}$	
		seq	Initial State $\rightarrow <$ Event Graph instance $>$		
		state	$\{0\}$		$P2$
		req	id	PWR	
			state	$\{5000, \ldots, 12000\}$	
			id	EN	
			state	$\{0\}$	
		seq	Initial State $\rightarrow <$ Event Graph instance $>$		
		state	$\{0\}$		$P3$
		req	id	PWR	
			state	$\{0, \ldots, 4999\}$	
			id	EN	
			state	$\{1\}$	
		seq	Initial State $\rightarrow <$ Event Graph instance $>$		
		state	$\{0\}$		$P4$
		req	id	PWR	
			state	$\{0, \ldots, 4999\}$	
			id	EN	
			state	$\{0\}$	
		seq	Initial State $\rightarrow <$ Event Graph instance $>$		
	set	s $\rightarrow$ VOUT_COMMAND(s)			

Table 5.10: A Producer instance for the MAX15301

5.4.3 Summary

In this section, we have discussed how producers are represented in our platform model. We have motivated their input/output pin-based descriptions with observations made in section 5.4. We have encountered the State Possibility structure, which condenses all information on and requirements of a specific State Space an output conductor can adopt. We have yet to define the nature of the sequence requirements attribute (**seq**), which will be the topic of the next section.

As an example, we have developed a producer description for a MAX15301 voltage regulator. This process required us to make some assumptions about the platform's electrical ground and to decide whether to model the conductor connected to the EN pin as a logical signal.

5.5 Events

As mentioned in the previous section, we need to describe how the BMC can cause conductor state changes. This description must contain the specific commands the BMC must issue, for instance the setting of GPIO pins used as outputs (see section 3.4.2). However, the situation is slightly more complex than this: Some of the BMC's commands and some producers feature *context sensitive* behaviour.

A prime example for this are PMBus commands that do not have any effects if the receiving producer is powered-down. Therefore, the producer must be powered **before** we issue a PMBus command. Another example are producers like the ISL6334d [20]: if its VID inputs feature certain values during power-up, the ISL6334d will immediately power-down and needs to be power-cycled to become operational again. It is therefore vital that we only start powering this particular producer **after** ensuring that the VID inputs feature the correct values.

For this reason, a simple description of BMC commands is not sufficient. We need a formal way to describe which conductor changes must happen before or after other conductor changes; in other words, we need to specify dependencies between conductor changes.

To do so, the BMC requires descriptions of how each conductor state change can be triggered and when it completes. We achieve this by associating two events with every conductor: an *Initiate* event and a *Complete* event. The goal is to then to *expose* every conductor's events, to allow other conductors to define relations between their own change and changes of other conductors. This is done on the level of State Possibilities: they must define the Initiate and Complete events for the output conductor they are associated with, along with other event relations. We will discuss the format of these descriptions in detail in section 5.5.5.

5.5.1 Initiate and Complete events

Initiate and Complete events relate to conductor state changes and are thus always associated with a static management action. Ultimately, we use these events to *construct* said management action, namely in the context of the *sequence generation* mechanism. However, to precisely define the meaning these events should have, we first define them with respect to some *fixed* static management action.

To clearly distinguish between the more *constructive* definition of events that we will make use of later on, and this semantic definition, we name the two versions differently: *Initiate_semantic* and *Complete_semantic* in contrast to simply *Initiate* and *Complete*.

As mentioned above, our semantic definitions will rely on a fixed static management action. This action is defined by some command sequence C, that has the platform transition from stable state S to stable state S'. In this context, we define *Initiate_semantic* and *Complete_semantic* as follows:

Definition 31 (Initiate_semantic)**.** Let w be some platform conductor.
Initiate_semantic(w) is an index i such that for all $x \leq i$, if we only executed the prefix of C of length x, w would retain its initial stable state $S(w)$.

Definition 32 (Complete_semantic)**.** Let w be some platform conductor.
Complete_semantic(w) is an index j such that for all $y > j$, if we only executed the prefix of C of length y, w would adopt its new stable state $S'(w)$.

Note that for all conductors w the following holds: If $Initiate\_semantic(w) = i$ and $Complete\_semantic(w) = j$ is valid, then for any $0 \leq x \leq i$ and $j < y <$ len(C), *Initiate_semantic(w) = x* and *Complete_semantic(w) = y* are valid too. *Initiate_semantic(w) = 0* and *Complete_semantic(w) = len(C) - 1* are trivially valid. We will use the term *tight* to signify that *Initiate_semantic(w)* and *Complete_semantic(w)* are defined as the largest respectively smallest indices possible.

Consider the case where conductor w is *independent* of command sequence C, i.e. none of the commands in C have any effect on the stable state of w. In such a situation, we would find that in their tightest definitions, Initiate_semantic(w) = len(C) - 1 and Complete_semantic(w) = -1. For the sake of simplicity, we will redefine Initate_semantic(w) = Complete_semantic(w) = -1 in this case, to signify that both events have happened already. With this redefinition, it thus always holds that Initiate_semantic(w) $\leq$ Complete_semantic(w).

We illustrate these definitions with an example. For the sake of simplicity, let us consider a simple set of three GPIO pins P1, P2 and P3 that the BMC is using as outputs. We describe the commands that set pin Pi's voltage level to LOW (0) and HIGH (1) with Offi and Oni respectively. We will furthermore describe any stable states of these GPIO pins as a vector [S1, S2, S3], whereby Si $\in \{0, 1\}$ denotes the state of pin Pi.
Consider the following static management action:

Initial State	Command Sequence C	End State
[0, 0, 1]	Off3, Off1, On2, On3, Off1	[0, 1, 1]

Table 5.11: An example static management action

In this example, the tightest definition of each pin's Initiate and Complete event would be as follows:

Pin	Initiate_semantic	Complete_semantic
P1	-1	-1
P2	2	2
P3	0	3

Table 5.12: Initiate and Complete events for the above management action (table 5.11)

Despite the Off commands at indices 1 and 4 referencing pin P1, we claim that P1 is independent of C; this is because P1 is already in a LOW state and hence Off1 does not affect its state.

The only command referencing pin P2 is On2 at index 2. Since P2 is initially LOW, On2 will cause it to transition to a HIGH state.

Merely looking at the initial and end state, one would be tempted to think that P3 is unaffected by C. This is, however, not the case: If we were to execute a prefix of C of length 1 to 3, P3 would end up in a HIGH end state.

With our semantic event definitions in place, we can once more refine our perspective on static management:

5.5.2 Ideal Static Management

Recalling the example in table 5.11, we realise that the behaviour pin P3 is exhibiting during this management action is best avoided. Imagine a scenario where P3 is connected to the enable signal of the platform's main power supply: this would result in us flicking the lights of the entire platform off and on again, which would most likely be devastating for any attached consumers.

Looking at the Initiate and Complete events which we defined for this example (see table 5.12), we can see that they reflect P3's *misbehaviour*: Initiate_semantic(P3) is not equal to Complete_semantic(P3). Quite generally, we can make the following observation:

Observation 33. Let w be a conductor with initial stable state $s1$ and final stable state $s2$. *Initiate_semantic(w)* $\neq$ *Complete_semantic(w)* has one of the following implications:

1. The event definitions are not tight
2. conductor w transiently attempts to adopt a stable state $s3$ different from both $s1$ and $s2$, which would manifest itself if we executed a prefix of the command sequence of length between *Initiate_semantic(w)* + 1 and *Complete_semantic(w)*.

In the context of static management, the second case should be avoided. Generally, we cannot rely on our commands taking effect quickly enough that such a state $s3$ is not expressed, and it might very well be the case that state $s3$ jeopardises the correct operation of the attached consumers. Thus, we define an ideal static management action as follows:

Definition 34 (Ideal Static Management Action)**.** For any conductor w and for tight definitions of Complete and Initiate events, the following should hold in context of every *fixed* static management action:

Initiate_semantic(w) = Complete_semantic(w)

Thus, the complete state change of every conductor w boils down to either a single command or none, in the case where w is independent of the management action (and thus *Initiate_semantic(w) = Complete_semantic(w) = -1*). With this, we guarantee that any conductor state fluctuate as little as possible. In particular, any fluctuations can be solely attributed to the producers' implementation of the requested state changes and are therefore unavoidable.

The descriptions of these events and any necessary event dependencies are part of the producer descriptions. Consequently, how close we come to achieving this ideal vision of static management depends on the precision and completeness of the provided descriptions. Before we discuss the structures our model uses to capture event descriptions, we need to shift our perspective on events from fixed command sequences towards the *construction* of command sequences:

5.5.3 Constructive Events

As mentioned in section 5.5.1, we will make use of Initiate and Complete events to construct correct static management actions. Of course, our semantic definition of these events as fixed indices in a known command sequence is not quite sufficient for this purpose. We are therefore in need of a more general and flexible definition.

On a producer level, Initiate and Complete events are tied to a specific State Possibility P. Conceptually, their definition must therefore correctly handle all static management actions that lead to stable platform state that adheres to P. We will formally define what *adhere* means in section 5.9.1, but conceptually it just means that all conductors referenced by P are in a state as dictated by P.

More concretely, let w be a platform conductor connected to a producer's output pin whose description contains some State Possibility P. Let S' be some stable platform state that adheres to P and let S be an arbitrary initial platform state. We can then define *Initiate(w)* as follows:

Definition 35 (Initiate)**.** In every *fixed* command sequence C that has our platform transition from S to S' and that observes all event restrictions imposed by P, *Initiate(w)* defines or identifies a single command c with index i, such that *Initiate_semantic(w)* $= i$ is valid.

It would be perfectly reasonable to define Complete events in an analogous fashion. However, for the purposes of correctly constructing a command sequence this would not quite suffice. Recalling our definition of ideal static management (definition 34), we would hope to identify the same command c, and thus Initiate(w) would be indistinguishable from Complete(w). However, as mentioned in the beginning of the events discussion, we wish to use these events to express relations between conductor changes. Initiate and Complete events must thus *timing-wise* form upper and lower bounds for the interval during which the state change of w is happening. More formally, the following should hold (see figure 5.5 for a visual representation):

Observation 36. Let A be a static management action the BMC performed and let $\mathtt{T}$ be a function that applied to an event, returns the precise point in time at which said event happened during action A. Let furthermore w be an arbitrary platform conductor and *Change(w)* denote the time interval during which w's state was changing because of A. Then:

$$\mathtt{T}(\text{Initiate(w)}) \leq \text{Change(w)} \leq \mathtt{T}(\text{Complete(w)})$$

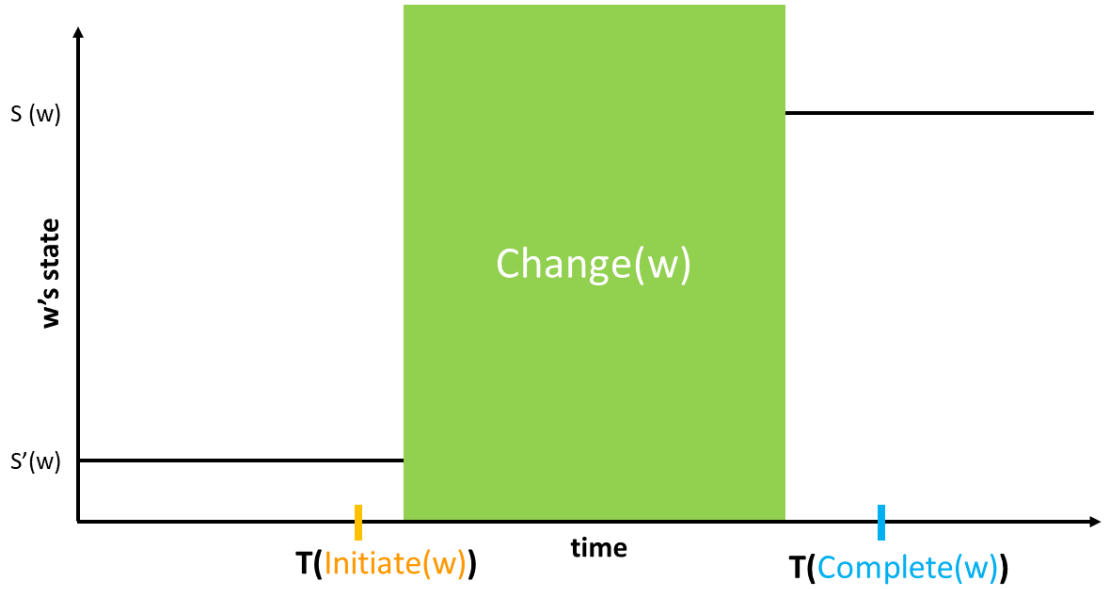


Figure 5.5: The timing relation between Initiate and Complete events

To accomplish this, our Complete events must act as *points of synchronisation*. More concretely, a *Complete(w)* event must be tied to a blocking monitoring action performed by the BMC that repeatedly queries the state s of conductor w until it finds that $s = S'(w)$. Thus, with S, S' and P as defined for the Initiate event definition:

Definition 37 (Complete)**.** In every *fixed* command sequence C that has our platform transition from S to S' and that observes all event restrictions imposed by P, *Complete(w)* defines a synchronisation action c on conductor w with target $S'(w)$, whereby index i of c is such that *Complete_semantic(w)* $= i$ is valid.

Remark. With the condition *Complete_semantic(w)* $= i$ being valid, we ensure that the blocking monitoring action will eventually complete.

With this definition, the *Complete(w)* event testifies that the change to conductor w has measurably happened. Thus, our desired timing relations as detailed in observation 36 holds.

With our working definitions of Initiate and Complete events in place, we can now discuss how we can describe relations between conductor changes by placing restrictions on the sequence in which certain events should happen.

5.5.4 Event Relations

As mentioned above, we will allow producer models to formulate restrictions on the sequence in which Initiate and Complete events should occur. In this section, we will

explore the possible restrictions together and evaluate their consequences and effectiveness. In order to do this as clearly as possible, we first introduce a schematic illustration of events and restrictions.

As the term *Event Graph* used in section 5.4.2 suggests, this illustration will be in the form of a directed graph. Intuitively, our nodes symbolize events and the directed edges will specify a $\leq$ partial order: a directed edge from event B to event A (see figure 5.6) means that $\mathtt{T}$(event A) $\leq$ $\mathtt{T}$(event B). As in the last section, function $\mathtt{T}$ applied to an event returns the point in time at which said happened.

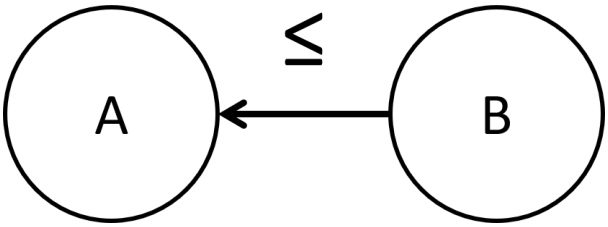


Figure 5.6: An example graph involving events A and B, including the edge interpretation as $\leq$.

Recalling observation 36, it follows that for every conductor w, T(Initiate(w)) $\leq$ T(Complete(w)), schematically:

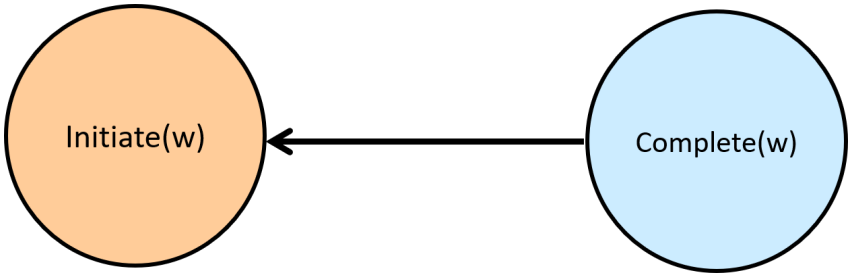


Figure 5.7: The timing relation between Initiate and Complete events as a graph

For the purposes of this discussion, we will adopt the perspective of a specific platform conductor w. We will attempt to describe how some event E concerning w (for instance Initiate(w) or Complete(w)) relates to another conductor u with u $\neq$ w. From the perspective of conductor w, the events Initiate(u) and Complete(u) are by themselves not really of interest, since only the BMC, u's producer and other producers offering monitoring functionalities on conductor u might be aware of them, but not conductor w. What can be of import, however, is when Change(u) is happening relative to event E. As indicated by observation 36, defining restrictions between E and Complete(u) or Initiate(u) *can* enforce certain relations between E and Change(u).

Generally, we can define four different restrictions between E and Complete(u) or Initiate(u), which we have depicted in figure 5.8.

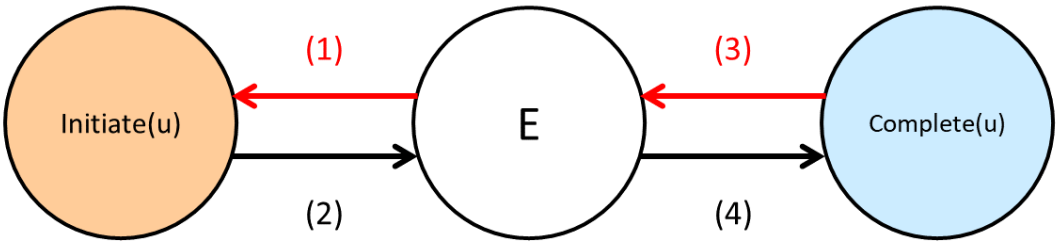


Figure 5.8: All possible restrictions we can specify between event E and Complete(u) and Initiate(u)

We will discuss the implications of each of these restrictions in turn:

1. this particular restriction is not very useful: T(Initiate(u)) $\leq$ T(E) does not imply any ordering of T(E) and Change(u)

2. this restriction ensures that event E happens **before** Change(u), since

$$T(\text{E}) \leq T(\text{Initiate(u)}) \leq \text{Change(u)} \leq T(\text{Complete(u)})$$

3. as in the first case, the restriction T(E) $\leq$ T(Complete(u)) does not imply anything about the relation of T(E) and Change(u).

4. this restriction ensures that event E happens **after** event Change(u):

$$T(\text{Initiate(u)}) \leq \text{Change(u)} \leq T(\text{Complete(u)}) \leq T(\text{E})$$

We have marked the two restrictions that do not make a lot of sense with red in figure 5.8. As can be seen quite nicely, only one arrow direction is sensible. More concretely, it only makes sense to force *Complete* events to happen *before* event E and *Initiate* events to happen *after* event E. If we were to omit all nonsensical restrictions, we would thus eliminate any reflexivity present in our event graph. For this reason, we enforce our partial ordering to be strict ($<$), and hence if there is an event sequence that observes all restrictions, the corresponding graph is a directed acyclic graph (DAG).

Since by definition, our Initiate and Complete events directly correspond to BMC commands, this implies that the BMC can issue commands sequentially. This is fortunate,

since even if the issuing of two commands at the exact some time were sensible, the I^2C bus protocol is inherently sequential, so it might not be technically feasible. In order to avoid confusion, we again present the relation picture above, but this time with the correct strict relation:

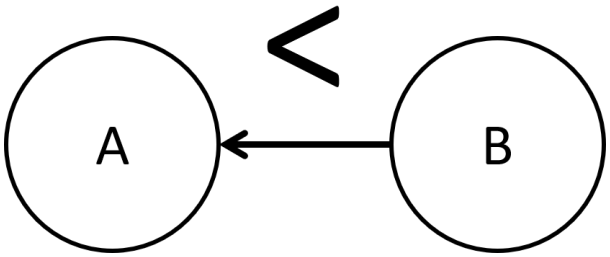


Figure 5.9: an example graph involving events A and B, including the **correct** edge interpretation as $<$.

5.5.5 Sequence Requirements

In this section, we discuss how our model represents event descriptions and restrictions. On a structural level, this representation is provided by the **seq** attribute defined by the State Possibility structure presented in section 5.4.2, which features the semantic structure *Initial State* $\rightarrow$ *Event Graph*.

The examples presented in the next section will make the need for the dependence on the *initial platform state* more clear. Since the precise format in which this initial state is expressed is irrelevant for our purposes, we focus our efforts on the Event Graph structure. Schematically, said is represented as follows:

Event Graph		
init	Initiate event description	*Initiate Event* | Happened
bI	before Initiate	{*string*}
aI	after Initiate	{*string*}
bC	before Complete	{*string*}
aC	after Complete	{*string*}
req	Happened Requirements	{*(string, string)*}

Table 5.13: The Event Graph structure

Note that for any conductor W, the natural relation Initiate(W) $\leftarrow$ Complete(W) is considered a *law of nature* and therefore does not need to be specified explicitly by an Event Graph structure.

Studying this representation closely, one might wonder why this schematic representation lacks a Complete event description. This is due to our *ideal* static management

vision that we have established in definition 5.5.2: if we manage to adhere to said, Initiate_semantic(W) = Complete_semantic(W) for all conductors W. Translating this to the constructive Initiate and Complete event definitions we are making use of, this means that the inherent relation Initiate(W) ← Complete(W) is sufficient to describe Complete(w) correctly (see definition 37). If a platform description does not adhere to this ideal case, the **aI** and **bC** attributes can be used to work around the lacking explicit Complete event description.

The **init** attribute defines the character of the Event Graph structure. It will either take the form of an Initiate Event structure or will be set to **Happened**, to signify that the state of the corresponding conductor will remain unchanged by this static management action and therefore the corresponding Complete and Initiate events have taken place in the past already. In order to ensure that the conductors state remains unchanged, additional requirements can be specified using the **req** attribute. The bI, aI, bC, aC attributes remain unused in this case. We will dedicate the next subsection to the **Initiate Event** structure.

The semantic meaning of the **bI, aI, bC and aC** attributes is straightforward: They allow for the specification of event restrictions as we have seen in the previous section.

Let Out be the Output structure this Event Graph structure belongs to. Then, in the notation used in the last section, conductor w corresponds to the conductor defined by Out and event E is Initiate(Out) for attributes bI, aI and Complete(Out) for bC, aC respectively.

The type of event that is sensible to reference is implied by the attribute. bI, for instance, allows to specify events that should happen *before* Initiate(OUT) and should thus only refer to Complete(u) events of other conductors u. It is hence sufficient to only specify these conductors u, which is precisely what is accomplished by the set of strings that comprises the syntactic structure of these attributes. Similar to State Requirements, these strings may only reference conductors that are connected to pins the producer defines: Again recalling our original producer definition, the following must hold:

$$\forall\ \mathrm{id} \in (\mathrm{bS} \cup \mathrm{aS} \cup \mathrm{bC} \cup \mathrm{aC}).\ id \in \{\mathrm{IN1}, \ldots, \mathrm{INn}, \mathrm{OUT1}, \ldots \mathrm{OUTm}\}$$

As mentioned above already, the **req** attribute is used only if init is set to Happened. In this case, we can specify an additional set of requirements in the following format: An element (A, B) of req specifies that the change to conductor A must happen **before** the change to conductor B. This thus corresponds to the following event restriction:

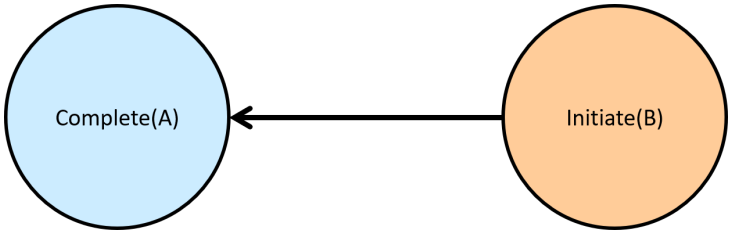


Figure 5.10: Graphical representation of the req attribute

As is the case with the aI, bI, etc. attributes, req may only reference conductors that are connected to pins the producer defines. Thus:

$$\forall\ (\mathrm{A},\ \mathrm{B}) \in \mathrm{req}.\ (A \cup B) \subseteq \{\mathrm{IN1}, \ldots, \mathrm{INn}, \mathrm{OUT1}, \ldots \mathrm{OUTm}\}$$

5.5.6 Initiate Events

As already hinted at by our definition of Initiate events (see definition 35), there exist two main categories of Initiate events: **Explicit** Initiate events that *define* a command and **Implicit** Initiate events that *identify* a command. More concretely, we define these two types as follows:

Let w be a platform conductor. We refer to Initiate(w) as an **explicit** event if w's state change is triggered by a command that directly targets conductor w. In our model, this command directly corresponds to the **set** attribute defined by w's output conductor. An example for this would be conductors attached to GPIO pins the BMC is using as output. The BMC can directly set such conductor's state to high or low using a corresponding command.

We call Initiate(w) **implicit** if the action associated with it directly updates some other conductor w', w $\neq$ w'. The state of conductor w then changes as a consequence of the update to w'. This would for instance be the case in the following situation: Consider a MAX15301 that is currently regulating its output voltage to 5V. Let w be the conductor connected to its OUT pin and w' the conductor connected to its EN pin and a GPIO pin of the BMC (see figure 5.11). If now the BMC sets the logical level of w' to LOW (0), this will cause the MAX15301 to stop regulating the voltage across w, which will as a result drop from 5V to 0V.

The first attribute of an Initiate Event structure identifies the corresponding event as explicit or implicit.

Initiate Event		
type	Initiate event type	$\in$ {Explicit, Implicit}
sub	Implicit event type	$\in$ {First, Last}
events	Related events	$\in \mathbb{P}(String)$

Table 5.14: The Initiate Event structure

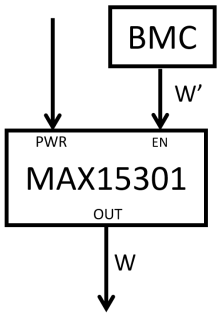


Figure 5.11: Situation used to provide an Implicit Initate Event example

If the event **type** is explicit, all further attributes are irrelevant; in practise, we will not draw the remaining attributes for explicit Initiate Event instances. Otherwise, they have the following meaning:

The **events** attribute identifies a set of conductors, whose Initiate events collectively define the implicit event described by this structure. As always, the local view a producer has on the platform only allows this attribute to reference conductors connected to other pins the producer defines:

$$\forall \text{ E} \in \text{events}. \ E \in \{\text{IN1}, \ldots, \text{INn}, \text{OUT1}, \ldots \text{OUTm}\}$$

The implicit events that need to be described for producers on the Enzian can all be further classified into two distinct subcategories: **First and Last** events. These subcategories define how the events referenced by the **events** attribute must be interpreted:

A **First** Implicit Initiate event is one where any of the referenced events by itself is sufficient to trigger the Initiate event described by this structure. Thus, this type of Implicit event is **resolved** to the time-wise *first* event to happen of the events referenced in the **events** attribute. An example for this is the last State Possibility instance described by our sample producer MAX15301:

state	{0}	
req	id	PWR
	state	{0, . . . , 4999}
	id	EN
	state	{0}
seq	< Event Graph instance >	

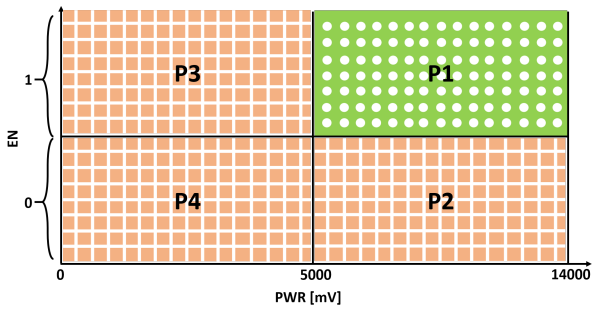
Table 5.15: Last State Possibility instance defined by OUT output pin of a MAX15301

As indicated by the other State Possibilities, both the EN signal being disabled as well as PWR dropping below 5V by themselves would be sufficient to cause the voltage across OUT to drop to zero.

A **Last** Implicit Initiate event is one where the combination of the referenced events triggers the Initiate event described by this structure. An example could be a producer that regulates its output to a fixed, statically defined voltage and will start to do so as soon as all its supply voltages and enable signals are online. Consequently, a Last Implicit event is resolved to the time-wise *last* of the referenced events that happens.

5.5.7 Sequence requirements for the MAX15301

In this section, we will discuss how we can describe adequate sequence requirements for the MAX15301. Our discussion will be driven by the following visualisation, which we had introduced in section 5.4.2:



Instead of exhaustively describing all sequence requirements the MAX15301 must specify for its OUT output, we will focus on a select few descriptions, which exemplify the use of all attributes defined by the *Event Graph* structure. Furthermore, we will specify every description in a fashion that provided that the event descriptions of the conductors connected to inputs PWR and EN adhere to *ideal static management* (see definition 34), our own descriptions for OUT will adhere to ideal static management too. This will provide us with a few insights about the quirks and general feasibility of ideal static management.

With the assumption that EN and PWR adhere to ideal static management, it follows that any of their state transitions are, if sensibly implemented by the corresponding producer, *smooth* and happen in a *direct* manner without unnecessary detours to other stable states. In our discussion, we will occasionally refer to this *smoothness* assumption.

Recall that the *seq* attribute to which our event descriptions are tied on a State Possibility level (see section 5.4.2) is of the form *Initial State* $\rightarrow$ *Event Graph*, i.e. depends on the Initial State from which we wish to transition to the new State Possibility. We will therefore specify which transition our description is attempting to accomplish in the paragraph headers in the following format:

Pi $\rightarrow$ Pj

Whereby Pi denotes the State Possibility instance corresponding to our Initial State and Pj the State Possibility instance we wish to achieve. Thus, the description would be included in Pj.

P1 → P4 We have already used this particular transition as a good example of when a *First* Initiate event might occur. Because of our smoothness assumption, we do not need to impose any further event requirements to adhere to ideal static management: No matter how we leave P1 (the green, dotted quadrant), as long as long as the state changes of EN and PWR do not feature any detours, the transition of OUT from (0.5V - 5.252V) to 0V will be smooth.

P1 → P4 [Event Graph]		
init	type	Implicit
	sub	First
	events	{PWR, EN}

Table 5.16: The Event Graph instance for P1 → P4

P2 → P3 From an ideal static management perspective, this transition is pretty interesting. Because of our *smoothness assumption*, we can imagine three fundamental ways in which a transition from P2 to P3 could happen. We have visualised these in our level plot:

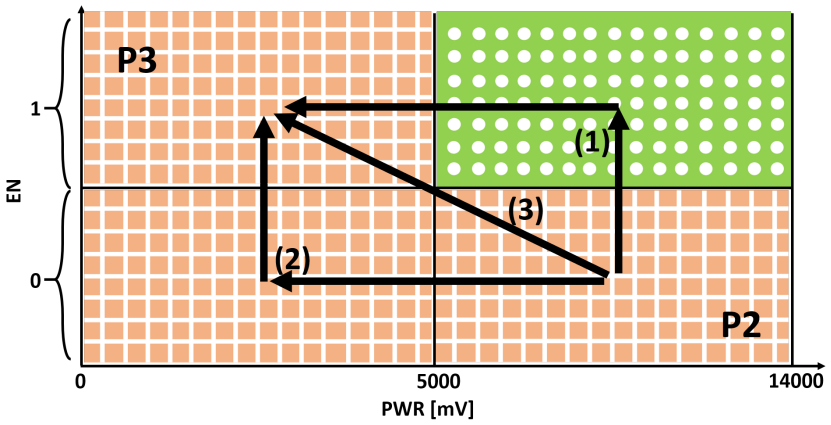


Figure 5.12: Possible transitions from P2 to P3 within the MAX15301's level plot

Since the state of OUT is 0V in both P2 and P3, according to our ideal management restrictions we must ensure that it remains at 0V during the whole transition. This restriction has the following consequences on our transitions:

1. This transition is inadequate, since we pass through the green, dotted quadrant (P1). Therefore, OUT might transiently adopt a state described by P1 (0.5 to 5.252 V).
2. This transition fulfils our requirement: our transition is completely contained in the red (grid pattern) quadrants and OUT will thus always feature 0V.
3. Technically, this transition is fine; however, for the transition to happen exactly like this, the changes of EN and PWR must happen simultaneously, which we cannot enforce with our event restrictions

Thus, the only viable transition corresponds to (2), which requires the change of PWR to happen before the change in EN. The corresponding Event Graph instance defines the Initiate and Complete events as *already happened*, while imposing the additional requirement that Complete(PWR) ← Initiate(EN):

P2 → P3 [Event Graph]	
init	Happened
req	{(PWR, EN)}

Table 5.17: The Event Graph instance for P2 → P3

P1 → P1 This corresponds to a transition from the green, dotted quadrant to itself, which designates an OUT state of 0.5V to 5.525V. To communicate to the MAX15301 which voltage it is supposed to provide, the PMBus command *VOUT_COMMAND* is used, which we have defined as the MAX15301's set attribute. Consequently, our Event Graph for this transition will feature an Explicit Initiate Event, since we require the BMC to issue said command.

Due to our smoothness assumption, PWR and EN will not change their state during this transition and thus no further event restrictions must be imposed. The resulting Event Graph looks as follows:

P1 → P1 [Event Graph]		
init	type	Explicit

Table 5.18: The Event Graph instance for P1 → P1

(P2, P3, P4) → P1 We must elaborate on the exact operation of the MAX15301 for our last example. As mentioned above, the VOUT_COMMAND is used to indicate to the MAX15301 which output voltage in the range (0.5V to 5.252) it is supposed to provide.

What happens, however, if we transition from P2, P3 or P4, where OUT features voltage 0V to P1 *without* specifying any voltage? In this case, the MAX15301 will regulate OUT to some *default voltage*. Our description of the MAX15301 does not provide us with any indication of what this default voltage might be. Consider the case where we want some output voltage V on OUT. Since we cannot determine if V corresponds to this default voltage, this has the following implications for any transition from (P2, P3, P4) to P1:

- We must always explicitly state which output voltage V we need, since we (naturally) cannot rely on V = default voltage. Thus, our Initiate Event must be explicit since we need the BMC to issue our VOUT_COMMAND
- In order to adhere to ideal static management, we must prevent the MAX15301 from regulating to its default voltage.

This furthermore has the following consequences for the circumstances during which our Explicit Initiate Event must happen:

1. The MAX15301 is only responsive to bus commands when powered. Consequently PWR must be in some state {5V - 14V}, i.e. we must be in quadrants P1 or P2 when Initiate(OUT) occurs

2. The Initiate event may not occur in the green quadrant (P1), since otherwise the MAX15301 might transiently regulate OUT to the default voltage.

It thus follows that our Initiate event must occur in the quadrant described by State Possibility instance P2. For initial states in P2 and P4 this could be visualised in our level plots as follows (with or without the white arrow, depending on whether we start from P2 or P4):

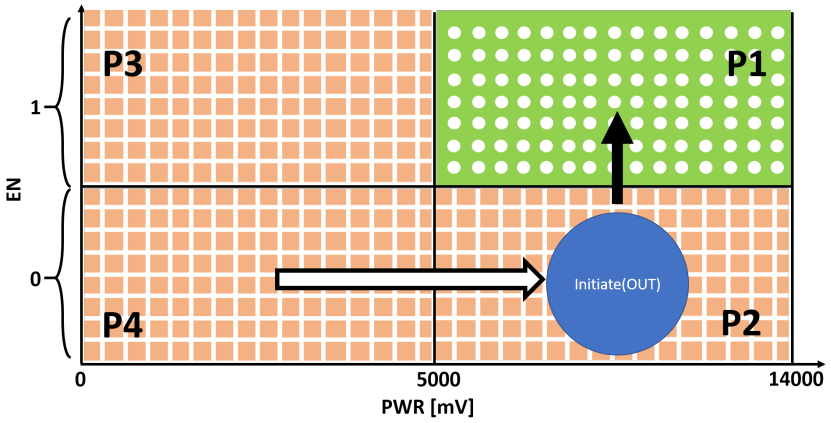


Figure 5.13: Possible transitions from P2 or P4 to P1 within the MAX15301's level plot

As can be seen in this visualisation, the corresponding Event Graphs need to enforce that the change to PWR completes *before* our Explicit Initiate event and the change to EN is initiated *after* our Initiate event and must complete *before* we attempt to verify that the OUT conductor has reached the desired state, i.e. *before* our Complete event. This results in the following Event Graph instance:

(P2, P4) → P1 [Event Graph]		
init	type	Explicit
bI	{PWR}	
aI	{EN}	
bC	{EN}	

Table 5.19: The Event Graph instance for P2, P4 → P1

Note that in this case, our Initiate event definition is not tight but instead a conservative approximation. The actual change of OUT is triggered by Initiate(EN), which we enforce to happen after Initiate(OUT) (using the **aI** attribute). However, this approximation does

not negatively affect our ability to find a viable sequence: Initiate(OUT) does not have any visible effect on the platform state apart from changing the internal configuration of the MAX15301.

For an initial state in P3, the situation is different. Since Initiate(OUT) must happen in quadrant P2, we would need to take the following path through our level plot to adhere to ideal static management:

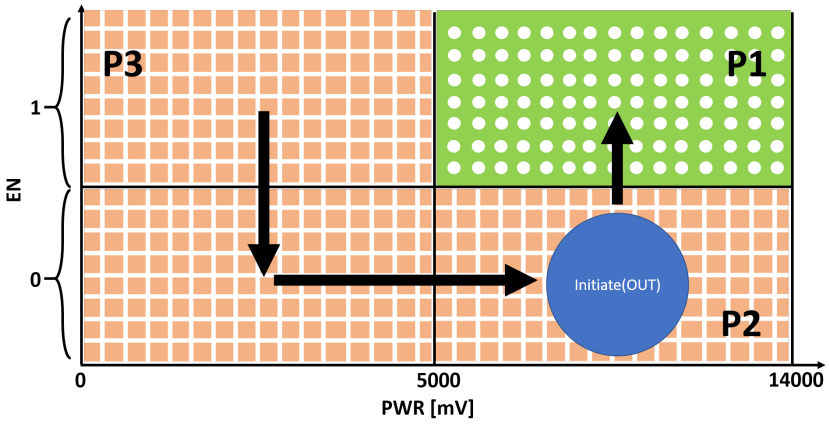


Figure 5.14: Possible transitions from P3 to P1 within the MAX15301's level plot

Studying this, we see that the behaviour of conductor EN does not adhere to ideal static management any longer: we change from state 1 to state 0 and back again. It follows that with our current way of modelling the MAX15301, it is **not possible** to transition from P3 to P1 in a fashion that adheres to ideal static management. One partial remedy for this is the modelling of default states, which we will discuss in section 5.6. With that, we can transition from P3 to P1 ideally, provided that the voltage we wish for OUT to adopt corresponds to the default voltage.

5.5.8 Summary

In this section, we have discussed how our model allows us to express restrictions on the timing relations between different conductor state changes. We have introduced the notion of Initiate and Complete events both in the context of a fixed command sequence as well as in a constructive sense. Based on these events, we have developed the concept of ideal static management that dictates that transitions between different conductor states should happen as smoothly as possible. We have then introduced the Event Graph structure our model uses to represent event restrictions. To conclude, we have developed some example Event Graphs for the MAX15301 and we have realised that in some situations, ideal static management is impossible to achieve.

5.6 Modelling the I²C bus

Until now, we have ignored I²C buses in our discussion of the platform model. We firstly wish to recall observation 24, which stated that the bus is not necessarily operational if any attached component is powered-off. This is something our platform model must not ignore: if an Initiate event of a conductor entails a bus command, we need to make sure that the bus is operational at that point in time, otherwise our generated command sequence will fail to accomplish the intended static management action.

In order to appropriately extend our platform model to correctly handle such buses, we must introduce some indicator of bus operability and we must extend our bus-compliant producer descriptions to take this indicator into account. We will discuss these extension in the next two subsections.

5.6.1 Operability indicator

Thinking about all the already established structures, modelling such an indicator as a logical output produced by an additional producer seems quite natural. We thus introduce a new producer instance that is concerned with the production of this logical output:

Bus [Producer]	
BUS	< Output instance >

Table 5.20: The Producer instance for an I²C bus

When attempting to decide how the referenced Output instance should look like, we very quickly realise the shortcomings of this approach: A bus is not a natural fit for a producer description. First of all, a bus does not feature a *fixed pin layout*; the State Possibilities of **BUS** are dynamically determined by the components attached to the bus. Secondly, and more significantly, a BUS logical signal is not quite a conductor in a static power management sense. We will discuss this issue in detail in section 5.6.4.

5.6.2 Producer extensions

Let P be a bus-compliant producer instance. For P to reference a bus indicator in its State Possibility instances, it must add this indicator as an additional input (**BUS**).

Furthermore, P must define the constraints it itself imposes on the indicator, namely a set of State Requirements that need to be imposed on P's input conductors to prevent P from pulling the bus low (**busreq**). Schematically, we add the following elements to P:

P [Producer]		
...		
BUS	amr	{0, 1}
	mon	
busreq	{<State Requirement instance>}	

Table 5.21: Adding the bus operability indicator to a bus-compliant Producer instance

It remains to discuss the integration of this bus indicator into the State Possibility instances defined by P. Let P_{BUS} be a State Possibility instance of P, whereby the sequence requirement function P_{BUS}.seq contains an argument value pair **(initial, event)**, such that the Event Graph **event** features an explicit Initiate Event that requires issuing a bus command.

We show how P_{BUS} can be modified for this particular argument value pair. Ignoring any other such pairs, P_{BUS} is of the following form, with the terms in **bold** serving as place holders for the actual values.

state	**state**		
req	**req**		
seq	initial →	init	**init**
		bI	**bI**
		aI	**aI**
		bC	**bC**
		aC	**aC**

Table 5.22: format of P_{BUS} when ignoring other argument value pairs next to **(initial, event)**

To take the operability of the bus into account, we must replace P_{BUS} with the following two new State Possibility instances:

state	**state**		
req	**req**		
	id	BUS	
	state	{1}	
seq	initial →	init	**init**
		bI	**bI** ∪ {BUS}
		aI	**aI**
		bC	**bC**
		aC	**aC**

state	**state**		
req	**req**		
	id	BUS	
	state	{0}	
seq	(initial ∧ BUS = {1})→	init	**init**
		bI	**bI**
		aI	**aI** ∪ {BUS}
		bC	**bC**
		aC	**aC**

Table 5.23: How to include bus operability in the above State Possibility instance

The first option, with differences to the original state possibilities coloured in green, ensures that the bus will transition to an operational state **before** the Initiate event (and thus before the pmbus command is issued).

The second option, with differences to P_{BUS} coloured in purple, has the bus transition to an inoperable state **after** the Initiate event. This requires ensuring that the bus was operational in the beginning and must thus modify initial accordingly. Note that in the case of an initial platform state corresponding to (initial $\wedge$ BUS $= \{0\}$), there exists no Event Graph structure that could ensure that the bus is operational when the bus command is issued.

5.6.3 Default States

When discussing examples of sequence requirements for the MAX15301 in section 5.5.7, we have already encountered an issue that could have been mitigated by extending our model to feature default states. Now that the bus being operational is not a given any more, we are *forced* to do so. Before we explain our reluctance, we need to give a more concrete definition of what exactly a default state is:

Definition 38 (Default State)**.** Let OUT be an output of a bus-compliant producer P. Let C be the last bus command that requested a certain state S of OUT and let T(Down) correspond to the most recent point in time at which P was powered-down. Furthermore, let D be the statically defined default value for OUT. Then, the default state of OUT is defined as follows:

$$\text{default(OUT)} = \begin{cases} S & \text{if T(C)} > \text{T(Down)} \\ D & \text{otherwise} \end{cases}$$

Or, less formally, as soon as a certain state is requested of OUT, said state is stored in the volatile memory of P. The default state is thus either this stored state or the value D if no stored state is available.

Clearly, this concept is in fundamental conflict with observation 13: The output states a producer can provide is generally independent of the previous input and command history. As already mentioned in section 5.4.2, by not allowing the *state* and *req* attributes of State Possibility to express any dependence on previous platform states, our producer descriptions are inherently based on this observation. Consequently, our basic platform model does not allow for the proper inclusion of default states.

This is entirely on purpose: extending our model to properly express default states would lead to a massive increase in descriptive complexity. Instead, we deal with this problem on the implementation level (see section 8.1.4).

5.6.4 Issues

As mentioned in subsection 5.6.1, a bus does not fit the concept of a producer very well. As a consequence, doing so severely restricts the platform behaviour our model can capture in the following two situations.

Issuing a bus command that causes the bus to be pulled low Consider the following schematic platform instance:

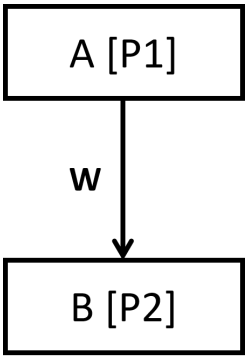


Figure 5.15: Platform instance illustrating the issue with bus commands that pull the bus low

Let us assume that both P1 and P2 are descriptions of bus-compliant producers and that A and B are connected to the same bus. Let us furthermore assume that the bus requirements P2 specifies require conductor W to supply at least X volts to prevent producer B from pulling the bus low. Consequently, our model does not allow the voltage of W to be set to a value Y $<$ X via a PMBus command to producer A.

To see why this is the case, we recall that with the producer extensions we described in section 5.6.2, the corresponding State Possibility instance for conductor W would be of the following format:

<table>
<tr><td>state</td><td colspan="3">state</td></tr>
<tr><td rowspan="3">req</td><td colspan="3">req</td></tr>
<tr><td>id</td><td colspan="2">BUS</td></tr>
<tr><td>state</td><td colspan="2">{0}</td></tr>
<tr><td rowspan="5">seq</td><td rowspan="5">(initial ∧ BUS = {1})→</td><td>init</td><td>init</td></tr>
<tr><td>bI</td><td>bI</td></tr>
<tr><td>aI</td><td>aI ∪ {BUS}</td></tr>
<tr><td>bC</td><td>bC</td></tr>
<tr><td>aC</td><td>aC</td></tr>
</table>

However, with W transitioning to state Y and therefore causing B to pull the bus low, Initiate(W) = Initiate(BUS). It follows that adhering to Initiate(W) ← Initiate(BUS) as required by attribute *aI* is impossible.

transiently available bus As already hinted at in subsection 5.6, modelling the bus operability as a logical platform conductor is not quite fitting. The reason for this is that our model is entirely focused on describing *stable* conductor states. The operability of a bus is not something that is required to assume a stable state. On the contrary, we are mainly interested in its value *while* we are performing static management actions. Consequently, it would in reality be sufficient for the bus to become operational only transiently while we issue the necessary commands, but our model cannot express this.

This is an issue in a situation like the following:

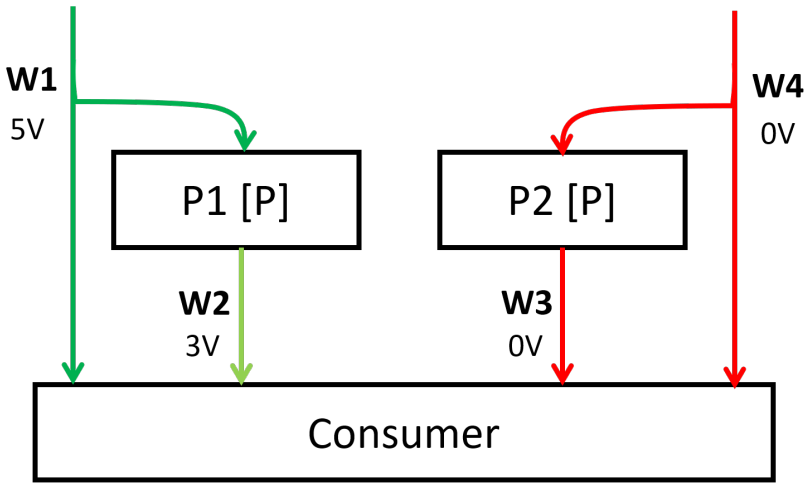


Figure 5.16: A Platform instance illustrating the issue of a transiently available bus

Let us assume that P is the description of a bus-compliant producer that pulls the bus low when powered off and that Producer instances P1 and P2 are connected to the same bus. Consequently, in the situation shown above, the bus would be pulled low by Producer instance P2.

The depicted Consumer instance now requests the following states for its input conductors: {W1: 0V, W2: 0V, W3: 3V, W4: 5V}, which exactly corresponds to the mirrored situation. Let us assume that 3V does not correspond to the statically configured default state of conductor W3, and must therefore be set by a bus command. In this case, there exists a transition that realises the requested change: We firstly change the state of conductor W4 to 5V. This causes the bus to become operational such that we can request 3V for conductor W3. We then change the state of W1 to 0V, which causes the bus to be pulled low again.

However, our model will not be able to find this transition because as already mentioned in section 5.6.2, no Event Graph exists that could ensure that the bus is transiently operational when we issue a command.

5.6.5 Summary

In this section, we have addressed the issue of bus operability that we had already identified in chapter 3. We have presented a possible solution that models the I^2C bus as an additional platform producer which outputs a logical signal indicating whether the bus is operational. In this process, we have realised that the nature of such a bus operability conductor fundamentally differs from a regular platform conductor. As a consequence, we have found our solution to be inadequate in two specific situations.

5.7 Consumer Description

Compared to producers, consumers are fairly easy to describe. Their general description layout is closely tied to the observations we made about consumers in section 3.3: We established that consumers demand certain states on their inputs and usually define a set of power states and transitions between said. We thus consider a consumer featuring n input pins named IN1, IN2, .., INn and m power states P1, P2, ..., Pm. Our consumer description will then be of the following format:

Consumer		
IN1	description of IN1	*Input*
	...	
INn	description of INn	*Input*
P1	description of P1	$\{State\ Requirement\}$
	...	
Pm	description of Pm	$\{State\ Requirement\}$
trans	power state transitions	$String \times String \rightarrow [\{State\ Requirement\}]$

Table 5.24: The Consumer structure

The syntactical structure of attributes **IN1, ..., INn** precisely correspond to the Input structures we have defined for producers.

Attributes **P1 to Pm** describe the consumer demands associated with the corresponding power state. The State Requirement structure used for this purpose has also already been defined in the context of producer descriptions. Similarly, we only allow the state requirements to reference Inputs that are defined by our consumer. Thus, for every power state attribute Pi the following must hold:

$$\forall \text{ p} \in \text{Pi}.\ p.id \in \{IN1, \ldots, INn\}$$

As hinted at by the syntactical structure, the **trans** attribute is supposed to be a function. It takes two strings as input and returns an ordered list of State Requirement sets. It describes the transition between any two different power states; let Pi and Pj be two arbitrary power states defined by our consumer. Then, trans(Pi, Pj) describes the transition **from** power state Pi **to** power state Pj. Once more, State Requirements are only allowed to refer to the inputs defined by the consumer. Thus:

$\forall$ x $\in$ {0, ..., length(trans(Pi, Pj)) - 1}, p $\in$ trans(Pi, Pj)[x] **.** p.id $\in$ {IN1, ..., INn}

That we model consumer demands using the rather simplistic State Requirement structure is in accordance with the assumptions made in section 4.1: Consumer demands are time-invariant and not associated with any real-time constraints.

5.8 Platform Description

As mentioned in section 5.2, we piece the model of our platform together using the individual producer and consumer descriptions and information on their composition. As such, every component present on the platform is some *named* instance of a producer or consumer description. We can then define the platform's conductors by indicating which output and input pins of which named instances it is connected to. Consider thus a platform PF with n producers P1 to Pn and m consumers C1 to Cm and p different conductors W1 to Wp. This platform's description is then of the following format:

Platform		
P1	description of P1	*Producer*
	...	
Pn	description of Pn	*Producer*
C1	description of C1	*Consumer*
	...	
Cm	description of Cm	*Consumer*
W1	connections to W1	*Connection*
	...	
Wp	connections to Wp	*Connection*

Table 5.25: The Platform structure

As detailed in observation 12, any conductor is an output to at most one producer. However, it may very well happen that a conductor is an input to several components in parallel. For this reason, our Connection structure, which specifies how the conductor in question is connected, has the following format:

Connection		
prod	specifies output producer	*string*
out	specifies output pin	*string*
in	specifies input pins	{(string, string)}

Table 5.26: The Connection structure

The **prod** and **out** attributes are quite self-explanatory: they identify the output pin the conductor is connected to, by specifying the name of the output producer instance and the output pin name. The following should therefore hold:

$\text{prod} \in \{\text{P1}, ..., \text{Pn}\} \land \text{type}(\text{PF.prod.out}) = \text{Output}$

The **in** attribute specifies the set of input pins: an element (A, B) of *in* indicates that our conductor is connected to the input pin with name B of the named producer instance A. Similar to the out and prod attributes, the following must hold:

$\forall\ (\text{A}, \text{B}) \in \text{in}.\ \text{type}(\text{PF.A.B}) = \text{Input}$

As an example, consider the schematic platform instance depicted by figure 5.17. With the pin names defined as indicated in the figure, the Connection instance of W5 would be of the form as shown by table 5.27.

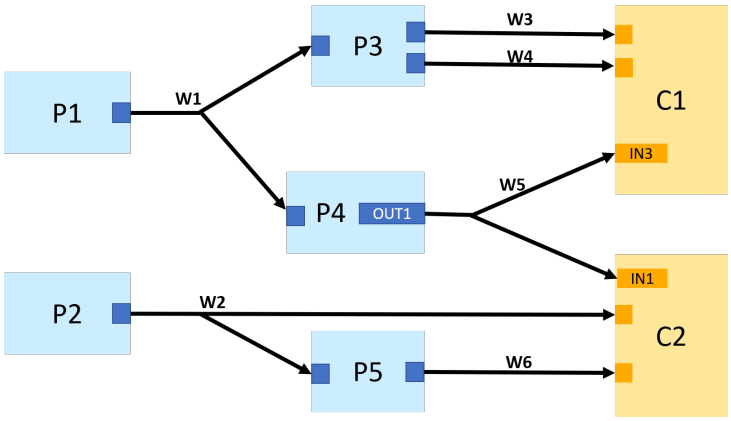


Figure 5.17: A schematic platform instance: Producer instances P1 to P5 and consumer instances C1 to C5 are drawn as coloured boxes and conductors W1 to W6 represented by black arrows. The pins defined by each component instance are drawn as smaller boxes in a darker shade, with Input pins drawn on the left and Output pins drawn on the right of component box.

W5 [Connection]	
prod	P4
out	OUT1
in	(C1, IN3), (C2, IN1)

Table 5.27: The connection instance for conductor W5 depicted in figure 5.17

Currently, our platform structure is unnecessarily bloated: Information on every conductor is distributed over multiple component descriptions. In the next subsection, we discuss how we can collect all information about a conductor in one Conductor structure. In subsection 5.8.2 we transform our Platform structure into a reduced and more convenient form using the defined Conductor structures.

5.8.1 Conductor description

As mentioned above, we combine all information on a conductor in a single Conductor structure:

Conductor		
mon	monitorng functionalties	$\{monitor\ action\}$
set	state set method	$Conductor\ State \rightarrow Command\ Action$
poss	state possibilities	$\{State\ Possibility\}$

Table 5.28: The Conductor structure

Before we discuss how this structure is composed, we must make one last, important observation. With our focus on **stable** conductor characteristics, namely voltage and frequency, we find ourselves in a fortunate situation: In parallel circuits, these characteristics remain the same across each parallel connection. We can therefore directly combine all the conductor state information given by the different input and output pins, without worrying how the conductor state might be **distributed**.

With this in mind, we can now discuss how we can reduce the redundant information present in our system. Let PF be a platform instance and W a Connection instance defined by PF. For the sake of simplicity, we define the following abbreviations:

$\mathbf{out} = PF.(W.prod).(W.out)$

Thus, **out** points to the unique Output instance the conductor described by W is connected to.

$$\mathbf{in} = \bigcup_{(A,B) \in PF.W.in} PF.A.B$$

The set **in** collects all Input instances the conductor described by W is connected to.

We define the Conductor instance of W as follows:

W [Conductor]	
mon	$\bigcup_{input \in in}$ input.mon
set	**out**.set
poss	$\bigcup_{p \in \mathbf{out}.\text{poss}}$ `Reduce`(p)

Table 5.29: The construction of a Conductor instance

The **mon** attribute simply collects all monitoring actions specified by the Input instances in **in**.

The **set** attribute corresponds to the set attribute defined by **out**.

The **poss** attribute is constructed from all State Possibility instances defined by **out**. We call a function `Reduce` on each such State Possibility instance P, which is supposed to transform P to include all necessary composition and conductor information. In particular, `Reduce` should:

- *crop* the state space of W described by *P.state* to adhere to all *absolute maximum ratings* defined by the input pins of W
- replace all mentions of pin names in P with the names of the connected conductors.

In order to define `Reduce` as precisely as possible without delving too much into the nested State Possibility structures, we define the following convenience function:

Let **prod** = PF.W.prod be the Producer instance that defines **out** and let **pins** = {IN1, ..., INn, OUT1, ..., OUTm} be the pinnames defined by **prod**. Then:

`Rename`: *Structure* → *Structure*

$$\textbf{Rename}(s) = \begin{cases} s & s \notin pins \\ w \mid (\textbf{prod}, s) \in \text{PF.w.in} \cup \{(\text{PF.w.prod}, \text{PF.w.out})\} & \text{otherwise} \end{cases}$$

Whenever a structure s references a pin defined by **prod**, `Rename` will replace s with the name of the unique conductor w connected to s. Of course, our platform specification might leave some pins unconnected. In this case, `Rename` will fail and we will have `Reduce` discard the entire State Possibility.

Thus, `Reduce` operates as follows:

`Reduce`: *State Possibility* → *State Possibility*

Reduce	P [State Possibility]	
	state	**state**
	req	**req**
	seq	**seq**

=

P [State Possibility]	
state	$\textbf{state} \cap \left(\bigcap_{input \in \textbf{in}} input.amr \right)$
req	`Rename`(**req**)
seq	`Rename`(**seq**)

Table 5.30: `Reduce` applied to a State Possibility instance

We conclude this section with a concrete example. Consider the schematic partial platform instance described by figure 5.18, as well as the corresponding pin instances provided by table 5.31:

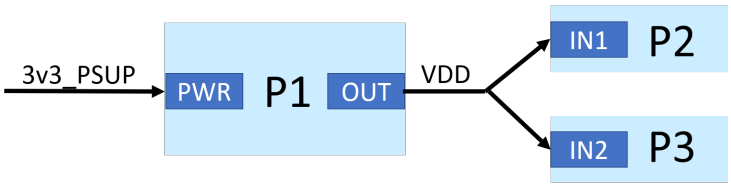


Figure 5.18: A partial Platform instance, used to illustrate the creation of a Conductor instance for conductor VDD

P1.OUT [Output]					
poss	state	{500, ..., 4000}			
	req	id	PWR		
		state	{5000, ..., 55000}		
	seq	Initial_State →	init	Implicit	
				First	
				{PWR}	
	...				
set	f3				

P2.IN1 [Input]	
amr	{-100, ..., 3000}
mon	mon1

P3.IN2 [Input]	
amr	{-300, ..., 2500}
mon	mon2

Table 5.31: Pin descriptions corresponding to the partial platform instance shown in figure 5.18

The Conductor instance we would construct for conductor VDD would in this case look as follows (changes compared to the description of P1.OUT are marked with blue):

VDD [Conductor]					
mon	{mon1, mon2}				
set	f3				
poss	state	{500, ..., 2500}			
	req	id	3v3_PSUP		
		state	{5000, ..., 55000}		
	seq	Initial_State →	init	Implicit	
				First	
				{3v3_PSUP}	
	...				

Table 5.32: A Conductor instance corresponding to the above pin and Platform instances

5.8.2 Reduced Platform Description

As already mentioned above, the goal of this section is to describe the transformation of the original Platform structure to a *Reduced Platform* structure. For this purpose, we will use of the Conductor structures whose construction we have discussed in the last section. But the Conductor structures alone are not quite sufficient:

Looking back at our original platform description, the only pieces of information that we have not merged into our Conductor structures are the power states and transitions defined by the platform's consumers. This makes sense: our Conductor structure includes all information related to the possible states said conductor can adopt and how we can have it transition between different states. The power states and transitions defined by consumers are conceptually on a higher level: they allow our management strategy to generate suitable consumer demands if the consumer does not do so actively (as discussed in section 3.4).

Since the Input structures defined by Consumers are already merged into the corresponding Conductor structures, our Reduced Platform does not need to retain said. We therefore introduce a *Reduced Consumer* structure that lacks these inputs:

Consumer		
IN1	description of IN1	*Input*
...		
INn	description of INn	*Input*
P1	description of P1	{*State Requirement*}
...		
Pm	description of Pm	{*State Requirement*}
trans	power state transitions	*String* × *String* → [{*State Requirement*}]

Reduced Consumer		
P1	description of P1	{*State Requirement*}
...		
Pm	description of Pm	{*State Requirement*}
trans	power state transitions	*String* × *String* → [{*State Requirement*}]

Table 5.33: The original Consumer structure compared to the Reduced Consumer structure

Since we described consumer power states and transitions in isolation, as was the case with all component descriptions, we need to perform a *renaming* pass on them too. With the `Rename` function as defined in the previous section, we thus arrive at the following reduction:

`Reduce`: *Consumer* → *Reduced Consumer*

Reduce
C [Consumer]	
IN1	**IN1**
...	
INn	**INn**
P1	**P1**
...	
Pm	**Pm**
trans	**trans**

=

C [Reduced Consumer]	
P1	`Rename`(**P1**)
...	
Pm	`Rename`(**Pm**)
trans	`Rename`(**trans**)

Table 5.34: `Reduce` applied to a Consumer instance

Our Reduced Platform will thus be of the following format:

Reduced Platform		
conductors	platform conductors	{W1, ..., Wp}
C1	description of C1	*Reduced Consumer*
...		
Cm	description of Cm	*Reduced Consumer*
W1	description of W1	*Conductor*
...		
Wp	description of Wp	*Conductor*

Table 5.35: The Reduced Platform structure

Whereby for the sake of accessibility, we have added the attribute **conductors** that returns a set of all conductors that are described by the platform.

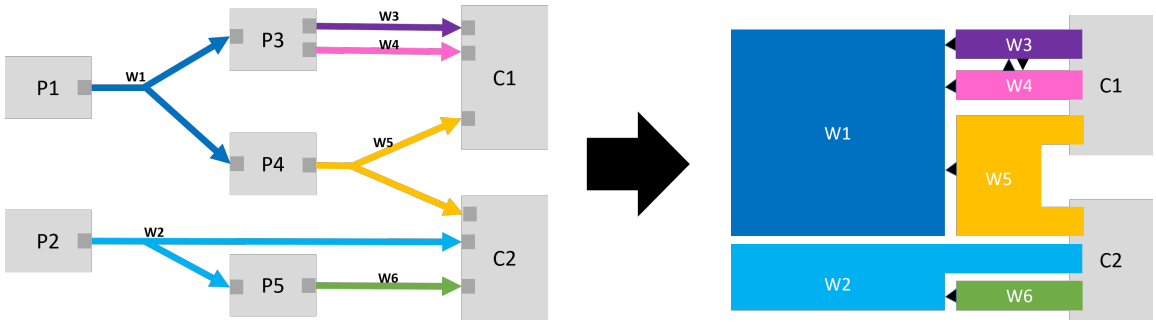


Table 5.36: A schematic illustration of the platform reduction process. The picture on the right features the original Platform instance with Producer instances P1 to P5, Consumer instances C1 and C2 and Connections W1 to W6. The corresponding Reduced platform instance is depicted on the right, with W1 to W6 now referring to Conductor and C1 and C2 to Reduced Consumer instances. The notation Wi ◁ Wj indicates that Wj may reference Wi.

To summarise the discussed platform reduction, we provide a schematic illustration of the process in table 5.36.

5.8.3 Summary

In this section, we have discussed how our model allows us to compose different component descriptions to a Platform instance. We have realised that the resulting construct includes a lot of redundancy. In order to remedy this, we have introduced an additional Conductor structure and described the transformation to a Reduced Platform instance.

5.8.4 A note on terminology

With this section, we conclude the structural description of our platform model. As a consequence, any subsequent mentions of the defined model structures such as Reduced Platform, State Possibility, Event Graph etc. will always refer to *instances* of these structures, not the structural definitions themselves. Always making this fact explicit by adding the term "instance" does not improve the readability of descriptions and argumentations, as the description of table 5.36 illustrates. We therefore define any subsequent reference of a model structure S to implicitly refer to an *instance of structure S*.

Furthermore, the transformation of a Platform to a Reduced Platform can be done autonomously by our model. Since the Reduced Platform structure is much more convenient, we will always assume that this transformation has been performed, and will not actively distinguish between the two structures any more in the remainder of this thesis.

5.9 Implications

5.9.1 Valid Platform States

The intended semantics of our platform model already give some indication about which platform states we consider valid. This section intends to formalise this idea; but before

we can properly reason about stable platform states, we must define them with respect to our platform representation.

Definition 39 (Platform State)**.** Let P be a Platform. A *Platform State* is a function that assigns a *Conductor State* to each Conductor defined by P. It is therefore a total function of the following format:
Platform State: P.conductors $\rightarrow \mathbb{Z} \times \mathbb{Z} \times \mathbb{N}$

Before we give any definitions of validity, let us recall how *valid* Platform States relate to our vision of Static Management. As we have formulated in definition 29, the stable platform states we generate must adhere to all *input constraints* to be considered valid. In our platform model, we have represented these input constraints using the *absolute maximum ratings* attributes of Input structures and the *State Possibilities* in Output structures. The reductions detailed in the last section resulted in the combination of these attributes into the **poss** attribute of the corresponding Conductor instance.

We therefore need to argue about validity in the context of Conductors. Since our platform's input constraints are distributed over all Conductors, a Platform State must be valid from the perspective of *every* individual Conductor to be valid in a global sense. Recalling the semantic meaning of *State Possibilities* as defined in section 5.4, we firstly formally define what it means for a Platform State S to be valid with respect to a State Possibility Poss:

Definition 40 (Valid_Poss)**.** Let W be a Conductor and Poss be a State Possibility defined by W. A Platform State S is considered to be valid with respect to Poss if the following predicate evaluates to true:

$$\text{Valid\_Poss(S, Poss, W)} = \text{S(W)} \in \text{Poss.state} \wedge \left(\bigwedge_{\text{req} \in \text{Poss.req}} \text{S(req.id)} \in \text{req.state} \right)$$

In other words, Valid_Possibility demands that S(W) agrees with Poss.state and all State Requirements defined by Poss are observed. Using this definition, we can now define the predicate Valid_Conductor(W, S) that indicates if Platform State S is considered valid from the perspective of Conductor W as follows:

Definition 41 (Valid_Conductor)**.** Let W be a Conductor of Platform P. The Platform State S is considered valid with respect to W if the following holds:
Valid_Conductor(W, S) = $\exists$ Poss $\in$ P.W.poss **.** Valid_Poss(S, Poss, W)

As discussed above, the global validity predicate Valid(P, S) of a Platform P and a corresponding Platform State S can thus be expressed as:

Definition 42 (Valid)**.** Let S be a Platform State. The validity of S with respect to Platform P is given by the following predicate:
Valid(P, S) = $\forall$ W $\in$ P.conductors**.** Valid_Conductor(W, S)

5.9.2 Valid Event Sequences

As discussed in section 5.5, it is important for the BMC to issue commands in the correct order. Our platform model employs the concept of Initiate and Complete events for this purpose: These events directly relate to commands the BMC is supposed to issue, and by placing appropriate restrictions on the sequence of events, our conductor descriptions can restrict the generated command sequence to a *correct* sequence.

In this section, we will develop a definition of what constitutes a valid event sequence based on the Event Graph structure we have introduced in subsection 5.5.5. Let hence **P** be a Platform and **Events**: *P.conductors* $\rightarrow$ *Event Graph* be an assignment of P's conductors to Event Graph structures. Let furthermore **S** be the event sequence whose validity we wish to determine.

We firstly observe that, depending on the nature of the assigned Event Graphs, Complete(W) and Initiate(W) should not be present in S for all conductors W. More concretely, if the Event Graph of W indicates that the event in question has already taken place, neither Complete(W) nor Initiate(W) should be appearing in the generated sequence. Neither should Initiate(W) occur in S if W's Event Graph declares it to be an implicit Initiate event; by definition of Implicit Initiate Events, Initiate(W) is supposed to *identify* another, explicit Initiate event. We summarise this observation by introducing the following two helper functions:

`Visible`: *String* $\times$ *Event Graph* $\rightarrow \mathbb{P}$(*Events*)

$$\texttt{Visible}(\text{W, E}) = \begin{cases} \emptyset & \text{E.init} = \text{Happened} \\ \{\text{Complete(W)}\} & \text{E.init.type} = \text{Implicit} \\ \{\text{Initiate(W), Complete(W)}\} & \text{E.init.type} = \text{Explicit} \end{cases} \tag{5.1}$$

The function **global_visible** applied to a Platform P and an Event Graph assignment *Events* returns the union of all visible terms:

`global_visible`: *Platform* $\times$ (*Conductor* $\rightarrow$ *Event Graph*) $\rightarrow \mathbb{P}$(*Events*)
`global_visible`(P, Events) = $\bigcup_{W \in P.conductors}$ Visible(W, Events(W))

With these terms, we can now define when we consider an event sequence S to be correctly formatted:

Definition 43 (Format)**.** Let P be a Platform, S be an event sequence and *Events* be an Event Graph assignment. We define the predicate **Format**, that decides if S is of the correct form as follows:

`Format`(P, Events, S) = S = permutation(global_visible(P, Events))

Meaning that the sequence S should be a permutation of the events that should be visible according to *Events*

In order to concisely define a valid predicate on a correctly formatted sequence S, we are in need of two more helper functions: First of all, `Edges`, which extracts all edges an Event Graph defines. This function precisely follows the definitions of the Event Graph structures, yet replaces the mentions of conductors with the corresponding events.

`Edges`: *String* $\times$ *Event Graph* $\to \mathbb{P}$(*Events* $\times$ *Events*)

$$\texttt{Edges}(\text{W, E}) = \begin{cases} \bigcup_{(a,b)\in E.req} \{(\text{Complete(a), Initiate(b)})\} & \text{E.init = Happened} \\ \Big(\{(Initiate(W), Complete(W))\} \cup & \\ \quad \bigcup_{bi\in bI} \{(\text{Complete(bI), Initiate(W)})\} \cup & \\ \quad \bigcup_{bc\in bC} \{(\text{Complete(bc), Complete(W)})\} \cup & \text{otherwise} \\ \quad \bigcup_{ai\in aI} \{(\text{Initiate(W), Initiate(ai)})\} \cup & \\ \quad \bigcup_{ac\in aC} \{(\text{Complete(W), Initiate(ac)})\} \Big) & \end{cases} \tag{5.2}$$

Secondly, the function `global_edges` applied to a Platform P and an Event Graph assignment *Events* returns the union of all edges defined by *Events*:

`global_edges`: *Platform* $\times$ (*Conductor* $\to$ *Event Graph*) $\to \mathbb{P}$(*Events* $\times$ *Events*)
`global_edges`(P, Events) = $\bigcup_{W\in P.conductors}$ Edges(W, Events(W))

Last but not least, we are in need of a function that helps us resolve implicit events and check whether or not our extracted dependency edges are respected. For this purpose, we resort to sequence indices. The connection between indices and our usual timing notation is clear: if two events E1 and E2 feature indices i1 and i2 in S with i1 $<$ i2, then T(E1) $<$ T(E2) and consequently E1 $\leftarrow$ E2. We define the function **index** as follows:

Let *Events* be an Event assignment, S a correctly formatted event sequence. We define the function index based on whether the event passed to it is a Complete or an Initiate event:

index(P, Events, S, Complete(W)) =

$$\begin{cases} i \text{ such that S[i] = Complete(W)} & \text{if Complete(W)} \in \text{global\_visible(P, Events)} \\ -1 & \text{otherwise} \end{cases}$$

index(P, Events, S, Initiate(W)) =

$$\begin{cases} i \text{ such that S[i] = Initiate(W)} & \text{if Initiate(W)} \in \text{global\_visible(P, Events)} \\ -1 & \text{if Events(W).init = Happened} \\ \min(\{\text{index(P, Events, S, Initiate(e))} \mid e \in \text{Events(W).init.events}\}) & \text{if Events(W).init.sub = First} \\ \max(\{\text{index(P, Events, S, Initiate(e))} \mid e \in \text{Events(W).init.events}\}) & \text{if Events(W).init.sub = Last} \end{cases}$$

Comparing the above definitions to our definition of visible events, it is apparent that we only assign -1 to an event if said has already happened. Furthermore, as our informal definitions of Last and First Implicit events dictated, we resolve them to the index of the last respectively first Initiate Event in their referenced events. Note that this could very well be an event that has already happened.

With all this preparation in place, we can finally define the valid predicate on event sequences:

Definition 44 (Valid_Seq)**.** Let P be a Platform, S be an event sequence and Events an event assignment of the form P.conductors → *Event Graph*. We define the **Valid_Seq** predicate that decides whether S correctly realises the Event Graphs mapped by Events as follows:

$$\begin{aligned} \textbf{Valid\_Seq}(\text{P, Events, S}) = \text{Format(P, Events, S)} \wedge \\ \Big(\forall\ (\text{a, b}) \in \text{global\_edges(P, Events)}. \\ \text{index(P, Events, S, a)} < \text{index(P, Events, S, b)}\Big) \end{aligned}$$

Note that with this definition, we treat cases where an implicit event Initiate(W) resolves to an already happened event conservatively: If Events(W) specifies any other conductor W2 in the bI attribute, by definition no valid event sequence exists, since -1 $\leq$ index(Complete(W2)) $\not<$ -1 = index(Initiate(W)). This makes sense; if two events are classified as *already happened*, we lack the required information to make any statement about the sequence in which they happened.

For this reason, it is highly recommended to prevent implicit events from resolving to already happened events by making use of the information about the initial platform states: recall that Event Graphs are obtained by passing the initial platform state to the function `seq`: *Initial State* → *Event Graph* defined by a State Possibility.

5.9.3 Valid Consumer Transitions

We have given each consumer the opportunity to define different power states Pi and transitions between said. As a reminder, we have included the schematic Reduced Consumer structure that we have developed in section 5.8:

Reduced Consumer		
P1	description of P1	{*State Requirement*}
...		
Pm	description of Pm	{*State Requirement*}
trans	power state transitions	*String* × *String* → [{*State Requirement*}]

The attribute **trans** is a function that takes the names of two power states P1 and P2 and returns an ordered sequence of consumer demands that, if fulfilled in this sequence, implements the transition from P1 to P2.

As already discussed in our motivation for the Consumer Demand Generation in section 4.2.1, we may not be able to exactly perform the sequence trans(P1, P2). Multiple consumers might need to transition between power states simultaneously, so we must generate a sequence that corresponds to a *valid* interleaving of all transitions. In this section, we define what valid means in this context.

For the following discussion, we need to define what the *subset* relation means if applied to Consumer Demands C1 and C2:

$$\text{C1} \subseteq \text{C2} \iff \forall\ \text{req} \in \text{C1}.\ \exists\ \text{req2} \in \text{C2}.\ \text{req.id} = \text{req2.id} \land \text{req.state} \subseteq \text{req2.state}$$

Which is precisely what we would expect: C1 ⊆ C2 if C1 is more strict than C2, i.e. whenever Consumer Demands C2 should be fulfilled, it suffices to fulfil C1.

We first define when some sequence of consumer demands correctly implements some transition:

Definition 45 (Valid_trans)**.** Let C be some consumer and P1 and P2 arbitrary power states defined by C. Let S be a sequence of consumer demands. Then, S correctly implements trans(P1, P2), which we denote with **Valid_trans(C, P1, P2, S)** if the following holds:

There exists some strictly increasing sequence of indices i(0) < ··· < i(L-1) such that the following holds, whereby L = len(trans(P1, P2)):

$$\text{S[j]} \subseteq \begin{cases} \text{C.P1} & \text{if } 0 \leq \text{j} < \text{i(0)} \\ \text{C.P2} & \text{if i(L-1)} \leq \text{j} < \text{len(S)} \\ \text{trans(P1, P2)[i(x)]} & \text{if i(x)} \leq \text{j} < \text{i(x+1)} \end{cases} \tag{5.3}$$

Conceptually, the indices i(0) to i(L-1) identify a *subsequence I* of S that exactly corresponds to trans(P1, P2). With our condition we ensure that no index j that is not in i(0),

..., i(L-1) jeopardises the transition. Thus, from the perspective of consumer C, there is no difference between I and S.

With this definition, it follows naturally that an interleaving S of several consumer transitions is valid if it is valid from the perspective of every consumer C:

Definition 46 (Valid_interleaving)**.** Let **Trans** $\in \mathbb{P}(\text{String} \times \text{String} \times \text{String})$ be a set of descriptions of consumer transitions, i.e. (C, P1, P2) $\in$ Trans means that C should transition for power state P1 to P2. Then, the sequence of consumer demands S correctly implements Trans, denoted with **Valid_interleaving(Trans, S)** if the following holds:

$\forall$ (C, P1, P2) $\in$ Trans**.** Valid_trans(C, P1, P2, S)

We will make use of these definitions when discussing our Consumer Demands Generation Mechanism.

5.9.4 Summary

In this section, we have introduced three additional structures that are relevant in context of our model: Platform States, Event Sequences and Consumer Demand Sequences. For each of these constructs, we have formalised the notion of validity in context of a specific Platform. We will make use of the resulting predicates in the next chapter that is dedicated to our mechanisms.

Chapter 6

Mechanisms

In this chapter, we take a closer look at the mechanisms we need to build on top of our model. The discussion of each mechanism is structured into three parts:

1. A precise problem definition in context of our platform representation
2. A discussion of the general complexity of the problem
3. An abstract description of the algorithm employed to solve the problem

6.1 SAT

In this chapter, we will perform two different reductions to the Boolean Satisfiability problem. In this section, we introduce the terms and notation we use to represent such a SAT problem.

Quite generally, the Boolean Satisfiability problem asks if, given a set of n boolean variables V = {V0, ..., Vn-1} and a boolean formula F involving these variables, there exists a truth assignment of V that renders F true.

In the traditional SAT problem, the formula F is in conjunctive normal form, meaning that F is a *conjunction* of m clauses F[0], ..., F[m-1]:

$$F = F[0] \wedge F[1] \wedge \cdots \wedge F[m-1]$$

Each if these clauses F[i] is a *disjunction* of len(i) many literals F[i][0], ..., F[i][len(i)-1]:

$$F[i] = F[i][0] \vee F[i][1] \vee \cdots \vee F[i][len(i)-1]$$

As already hinted at by the notation, we wish to access individual literals by treating F like a 2-dimensional array. By definition, every literal corresponds to a single, optionally negated variable in V. We introduce four helper functions to assess the meaning of each individual literal or clause:

`value`: *Literal* $\rightarrow \{0, 1\}$
Is supposed to return the (integer) truth value the referenced variable should feature to render the literal True, i.e. 0 if the literal features a negation and 1 otherwise.

`var`: *Literal* $\rightarrow \{V0, \ldots, Vn-1\}$
Returns the name of the variable referenced by the literal.

`positive`: *Clause* $\rightarrow \mathbb{P}$(V)
positive(F[i]) = {var(F[i][j]) | value(F[i][j]) = 1}
Returns the set of all variables that are referenced as *positive* literals in the given clause

`negative`: *Clause* $\rightarrow \mathbb{P}$(V)
negative(F[i]) = {var(F[i][j]) | value(F[i][j]) = 0}
Returns the set of all variables that are referenced as *negative* literals in the given clause

We conclude this section with a small example. Consider the following boolean formula F:

$$\begin{array}{llll} (\neg V0 & \vee & V2) & \wedge \\ (\ V2 & \vee & V1) & \wedge \\ (\neg V0 & \vee & \neg V1) & \end{array}$$

In this case, `var`(F[0][1]) returns V2 and `value`(F[2][0]) evaluates to 0. Furthermore, `negative`(F[1]) = {} and `positive`(F[1]) = {V1, V2}

6.2 State Generation

6.2.1 Problem Definition

In section 4.2.1, we have defined the purpose of the State Generation mechanism as follows:

> The process of determining a new **stable platform state** (1) that observes the new **consumer demands** (2) without violating any **input constraints** (3)."

We must now establish how this goal, and in particular the three terms in bold, relate to our platform model:

When defining the nature of consumer descriptions in section 5.7, we have already discussed our model's representation of consumer demands: In accordance with the assumptions made in section 4.1, we declared consumer demands to be simplistic enough to be expressed by a set of *State Requirements*. Therefore, our State Generation mechanism expects to receive consumer demands C $\in \mathbb{P}$(State Requirements).

In section 5.9.1, we have formalised the meaning of terms (1) and (3): We have defined a Platform State to be an assignment of Conductor States to conductors defined by the platform. We have furthermore developed a Valid predicate for a Platform State that indicates if said observes all input constraints.

We can therefore formally express the purpose of the State Generation mechanisms as follows:

Definition 47 (State Generation Problem)**.** Given a Platform P and consumer demands $C \in \mathbb{P}$(State Requirements), the State Generation mechanism either returns a Platform State S with the following property:

$$\text{Valid(P, S)} \wedge \left(\bigwedge_{\text{req} \in \text{C}} \text{S(req.id)} \in \text{req.state} \right)$$

or outputs *None* if no such Platform State S exists.

There is, however, another term our State Generation Mechanism should ideally return: We recall that the Valid_Conductor predicate we defined in section 42 is based on the existence of a suitable State Possibility. Therefore, any mechanism that constructs a valid Platform State S must in some way or another *decide which State Possibilities* ***Poss*** *the Platform State S adheres to.* It makes sense for our Sequence Generation mechanism to explicitly return **Poss** for the following two reasons: Firstly, our model integrates the information required by the *Sequence Generation* into individual State Possibilities. Thus, if not made explicit, the Sequence Generation mechanism would need to recompute Poss. Secondly, it makes the reasoning about correctness of the reduction and the algorithm presented in the next section easier and more readable.

We therefore require the State Generation Mechanism to also return a function **Poss**: P.conductors $\rightarrow$ *State Possibility* that assigns to every conductor a State Possibility the constructed Platform State S adheres to. Note that the following property holds:

$$\begin{aligned} \forall\ \text{S, P}\textbf{.}\quad & \text{Valid(P, S)} \iff \\ & \exists\ \text{Poss}\textbf{.}\Big(\forall\ \text{W} \in \text{P.conductors}\textbf{.}\ \text{Poss(W)} \in \text{P.W.poss} \wedge \text{Valid\_Poss(S, Poss(W), W)}\Big) \end{aligned}$$

Because of this, we can arrive at the following, equivalent formulation of the State Generation problem:

Definition 48 (State Generation Problem)**.** Given a Platform P and consumer demands $C \in \mathbb{P}$(State Requirements), the State Generation mechanism either returns **(S, Poss)** with the following properties:

S is a Platform State
Poss is a function P.conductors $\rightarrow$ *State Possibility* and the following holds:

$$(\forall\ \text{W} \in \text{P.conductors}\textbf{.}\ \text{Poss(W)} \in \text{P.W.poss} \wedge \text{Valid\_Poss(S, Poss(W), W)})$$
$$\wedge \left(\bigwedge_{\text{req} \in \text{C}} \text{S(req.id)} \in \text{req.state} \right)$$

or outputs **None** if no such Platform State S exists.

In the following subsections, we will always refer to the second formulation of the State Generation Problem. Note that from a solution for our first problem formulation, the function Poss required by the second problem formulation can be computed in polynomial time. According to the analysis provided in the next section, the two formulations are therefore also equivalent in terms of computational complexity.

6.2.2 Problem Complexity

We claim that the State Generation problem is NP-hard. We will prove this claim with a reduction of the SAT problem to the State Generation Problem. For the sake of simplicity, we will abbreviate the State Generation Problem as SG in the following discussion.

Let (V, F) be an arbitrary instance of the SAT problem as defined in section 6.1 and let B be any algorithm that solves the SG problem. We are going to *efficiently* construct a Platform Plat and a set of consumer demands C such that the following relation between the return value of B called on (Plat, C) and our SAT problem instance exists:

$$\text{B(Plat, C)} = \begin{cases} (S, Poss) & \text{if SAT(V, F)} = \text{True} \\ None & \text{if SAT(V, F)} = \text{False} \end{cases} \tag{6.1}$$

Schematically, the Platform Plat we are going to model has the following format:

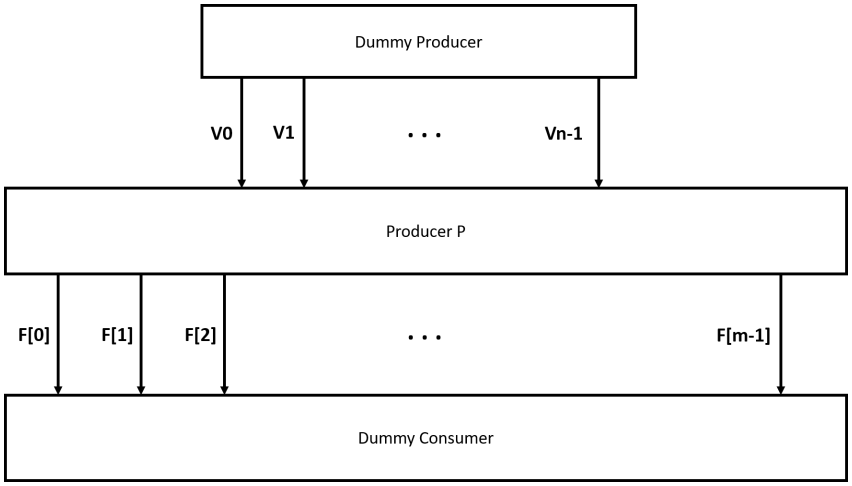


Figure 6.1: A schematic representation of Plat

All of Plat's conductors are transmitting logical signals. As the names given to the conductors imply, conductors V0 to Vn-1 will be symbolising the variables V of our SAT problem and conductors F[0] to F[m-1] the clauses of our boolean formula F.
The Producer P defines specific State Possibilities that capture the nature of the corresponding clause. More specifically, in the description of each of its outputs F[i], P defines a State Possibility for every literal in clause F[i] as follows:

state	{1}	
req	id	var(F[i][j])
	state	{value(F[i][j])}

Table 6.1: The j-th State Possibility P defines for its output F[i]

As their names imply, the Dummy Consumer simply serves as a point of connections for the outputs generated by P whereas the Dummy Producers generates the logical signals on V0 to Vn-1 without imposing any restrictions on said.

The corresponding Platform will thus be of the following format:

Plat [Reduced Platform]				
conductors	{V0, ..., Vn-1, F[0], ..., F[m-1]}			
V0	poss	state	{0, 1}	
...				
Vn-1	poss	state	{0, 1}	
F[0]	poss	state	{1}	
		req	id	var(F[0][0])
			state	{value(F[0][0])}
		...		
		state	{1}	
		req	id	var(F[0][len(0)-1])
			state	{value(F[0][len(0)-1])}
...				
F[m-1]	poss	state	{1}	
		req	id	var(F[m-1][0])
			state	{value(F[m-1][0])}
		...		
		state	{1}	
		req	id	var(F[m-1][len(m-1)-1])
			state	{value(F[m-1][len(m-1)-1])}

Table 6.2: The Platform Plat, any irrelevant attributes have been omitted

Clearly, this construction can be performed in polynomial time of the number n of variables in V and the number of literals $q = \sum_{i=0}^{m-1} len(i)$ in F: our Platform defines n + m many Conductors with a total of q + n many State Possibilities, each of which can be computed in constant time.

Now we just need to specify our consumer demands C, which we will set to $\emptyset$.

We now need to show that the desired correspondence defined above (equation 6.1) holds. To do so, we show that every solution (S, Poss) to SG(Plat, C) exactly corresponds to a truth assignment A of V that satisfies F. This implies that our algorithm B only returns None iff the formula F is not satisfiable, as required.

Let therefore (S, Poss) be a solution to SG(Plat, C). Let A be constructed as follows:

$\forall$ v $\in$ V**.** A(v) = S(v)

We claim that A is a truth assignment of V that satisfies formula F. We prove this by showing that in every clause of F at least one literal evaluates to 1 under A. Let therefore F[j] be an arbitrary clause of F. According to the definition of Poss, Poss(F[j]) $\in$ P.F[j].poss. Without loss of generality, let Poss(F[j]) be the i-th State Possibility of conductor F[j]. According to our construction, Poss(F[j]) is therefore of the following format:

state	{1}	
req	id	var(F[j][i])
	state	{value(F[j][i])}

Since furthermore Valid_Poss(S, Poss(F[j]), F[j]) must hold, it follows that in particular:

S(var(F[j][i])) $\in$ {value(F[j][i]} $\iff$ S(var(F[j][i])) = value(F[j][i])

It follows that A(var(F[j][i])) = value(F[j][i]), which by definition of the value function renders literal F[j][i] and thus clause F[j] true. Since F[j] was an arbitrary clause of F, it follows that indeed A satisfies formula F.

The other direction of the correspondence is very similar to the first, which is why we take the liberty to omit it. This concludes our discussion of the reduction of SAT to SG. Since SAT is known to be NP-hard, it follows that our Sequence Generation problem is NP-hard too.

6.2.3 Algorithm

The algorithm we propose to solve the State Generation problem is work-list and backtracking based: The algorithm operates with the following variables:

- **S**, a mapping P.conductors $\rightarrow$ *State Space*. S represents a more general version of a Platform State since it assigns State Spaces rather than individual Conductor States. At the end of the execution, we will construct the Platform State we are required to return from S.
- **Poss**, the function P.conductors $\rightarrow$ *State Possibility* the state mechanism is required to return.
- **Desired_States** $\in \mathbb{P}$(*StateRequirement*), which at any point during the execution will contain the State Requirements that we have not yet enforced in S.
- **Choices**, a stack where we store every option O that we have not yet pursued along with the execution states of S, Poss and Desired_States we need to revert to in order to pursue O. These options correspond to State Possibilities of conductors.

Initialisation

We initialise our variables as follows:

S	$\leftarrow$	(P.conductors $\rightarrow$ {None})
Poss	$\leftarrow$	(P.conductors $\rightarrow$ {None})
Choices	$\leftarrow$	$\perp$
Desired_States	$\leftarrow$	C

Thus, both S and Poss initially map every conductor to some placeholder value *None*, Choices is initialised with an empty stack and Desired_States with the given consumer demands C.

Procedure

With variable Desired_States holding all State Requirements that still need to be enforced, the termination of our procedure is naturally given by Desired_States becoming empty.

While Desired_States $\neq \emptyset$:
1: req $\leftarrow$ Desired_States.pop()
conductor $\leftarrow$ req.id
state $\leftarrow$ req.state

If S(conductor) is *None*:
Let Options be the set of State Possibilities of *conductor* that agree with *state*, i.e.:
2: Options $\leftarrow$ {p | p $\in$ P.conductor.poss $\wedge$ p.state $\cap$ state $\neq \emptyset$ }

If Options is empty:
Pop from Choices and revert
3: Else:
4: p $\leftarrow$ Options.pop()
5: Push remainder of Options and current execution state onto Choices
Poss(conductor) $\leftarrow$ p
6: S(conductor) $\leftarrow$ p.state $\cap$ state
7: Desired_States $\leftarrow$ Desired_States $\cup$ p.req

Else:
8: If S(conductor) agrees with state, i.e. S(conductor) $\cap$ state $\neq \emptyset$:
9: S(conductor) $\leftarrow$ S(conductor) $\cap$ state
Else:
Pop from Choices and revert Return (S, Poss)

If the procedure attempts to pop from Choices but finds the stack empty, it will terminate and return *None*.

Complete consumer demands

For the procedure described above to solve the State Generation problem correctly, we need to make an additional assumption about the nature of the consumer demands passed to it. Consider for instance the following Platform:

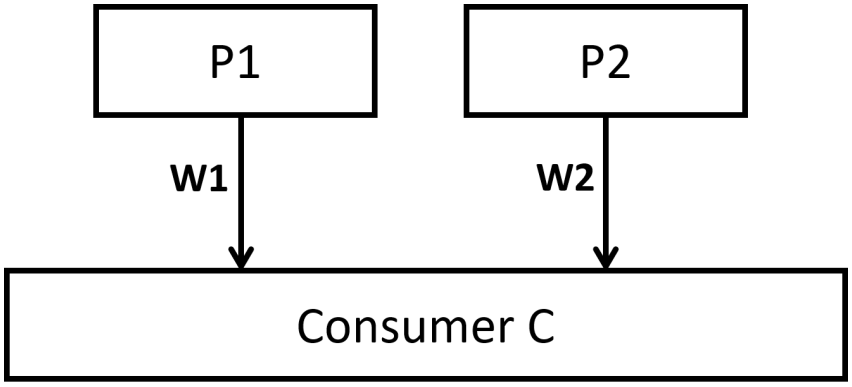


Figure 6.2: An illustration for the need of complete consumer demands

If the demands of consumer C were to only reference conductor W1, the above procedure would not choose a State Possibility for W2 and thus not assign any State Space to S(W2). We are therefore required to make the following assumption:

Assumption 49. The consumer demands passed to our State Generation procedure are sufficiently *complete*, meaning that either the procedure returns *None* or returns a *complete* State Space assignment S, meaning that: $\forall$ W $\in$ P.conductors**.** S(W) $\neq$ None

---

In our implementation, this assumption can be enforced by setting the flag *extend* to True, see section 8.2.4

Correctness

We firstly consider the correctness of the procedure for the case in which it does not terminate with *None*. We wish to argue that in this case, a valid solution to the State Generation problem as defined by definition 48 is given by **(S', Poss)**, whereby S' is constructed by assigning each conductor W an arbitrary element of S(W), and thus the following property holds:

$\forall$ W $\in$ P.conductors **.** S'(W) $\in$ S(W)

For this purpose, we proceed as follows: We firstly argue that such a construction of S' is possible, meaning that for all W, S(W) is not empty. Because of our assumption that for every conductor W, we perform at least one assignment to S(W) (assumption 49), it suffices to show that our procedure only ever assigns non-empty sets to S.

This is easy to see: the only assignments to S are performed on lines 6 and 9, and we ensure that the value we assign does not correspond to an empty set with checks on lines 2 and 8 respectively.

In order to continue our correctness argument, we need to make some assumption about the correctness of the revert mechanism, since we have not explicitly defined said in our procedure (deliberately, as it is not very interesting): We assume that our revert mechanism is implemented in a manner that the result any execution of the above procedure, no matter how unlucky, is equivalent to the result of an execution where we had an oracle that picked optimal state possibilities p (on line 4) for us and prevented us from needing to revert.

With this assumption, we are allowed to only consider such an optimal, oracle-guided execution, which we will do from now on.

We claim that every iteration of our while loop causes S to be *refined*. Looking at the control flow in our loop iterations, we see that the executions always ends up in the else-branch of the outermost IF statement if we have already assigned a State Space to S(conductor) in a previous iteration. In said branch, the value of the assignment to S(conductor) is S(conductor) $\cap$ state, and thus constitutes a refinement. This claim directly implies that no loop iteration ever undoes the work of a previous iteration. Furthermore, the fact that our refinement operator is the set intersection and therefore associative implies that the sequence in which we iterate over the elements in Desired_States does not matter.

Furthermore, every loop iteration succeeds in enforcing the State Requirement **req** that it has popped from the set of Desired_States on line 1. Since we consider an oracle-guided execution that never reverts, we find that on every execution path there is an assignment to S(conductor) $\equiv$ S(req.id) of a value that is an intersection between some other State Space and state $\equiv$ req.state. Together with the previous observation, this implies that once the procedure terminates, all State Requirements that have ever been in Desired_States have been successfully enforced on S.

With this observation, we can conclude that:

$$\left(\bigwedge_{\text{req} \in \text{C}} \text{S(req.id)} \subseteq \text{req.state} \right)$$

and consequently that our constructed Platform State S' observes the second property required by solutions to the State Generation Mechanism (Definition 48).What remains is to show the first property:

Since we assume that for every conductor W, S(W) has been assigned to at least once, it follows that we have executed the first Else-branch beginning on line 3 for every conductor W. Since we assume that we never had to revert, the State Possibility p chosen on line 4 corresponds to the value of Poss(W) at the end of the execution. It thus holds that for

every W, Poss(W) ∈ P.W.poss and S(W) ⊆ Poss(W).state. We furthermore add all of p's State Requirements to Desired_States on line 7, and with the refinement observations above this implies the first property defined by the State Generation Problem:

∀ W ∈ P.conductors**.** Poss(W) ∈ P.W.poss ∧ Valid_Poss(S', Poss(W), W)

We still need to argue that our procedure will only return *None* if no suitable Platform State exists. With our backtracking mechanism and the observation that the sequence in which we attempt to enforce State Requirements does not matter, our procedure performs an exhaustive search and should hence find a viable solution if one exists.

6.2.4 Finding all solutions

Note that with the backtracking state the above procedure keeps, we can easily modify it to return *all* possible combinations of State Possibilities. For this we simply store every solution the procedure returns and call it anew with its variables instantiated as given by the last execution state stored on *Choices*.

6.3 Sequence Generation

6.3.1 Problem Definition

We once more recall the initial problem definition we gave in our objectives section:

> Generating the sequence of BMC actions that will have the platform transition from the current stable state to the *new stable platform state*.

In our model, we have used Initiate and Complete events to reason about this sequence of actions. As discussed in section 5.5.6, an Initiate(W) event directly corresponds to issuing the Command Action of the associated conductor's *set* attribute. Similarly, a Complete(W) event corresponds to a synchronisation action involving the monitor functionalities W defines in its *mon* attribute. Because of this close correspondence between BMC actions and events, we will slightly reduce the above goal to only require the construction of a **sequence of events**. This makes our complexity argument in the next section a bit more intuitive.

Furthermore, since we had our State Generation mechanism collect the State Possibilities **Poss** from which we can directly extract our *seq* attributes and thus our Event Graph structures, our problem definition will make use of *Poss*. However, for the sake of simplicity, we take the liberty of pre-processing *Poss* in the following fashion:

We recall that attribute *seq* is of the form *Initial State* → *Event Graph*. Let P be our Platform and S' the *current* Stable State as defined in the above problem description. We construct a new function *Events* as follows:

Events: *P.conductors* $\rightarrow$ *Event Graph*
$\forall$ W $\in$ P.conductors**.** Events(W) = Poss(W).seq(S')

Recalling the **Valid_Seq** predicate (definition 44) we have developed in section 5.9.2, we arrive at the following problem definition:

Definition 50 (Sequence Generation)**.** Given a Platform *P* and a function *Events* of the form *P.conductors* $\rightarrow$ *Event Graph*, the Sequence Generation mechanism either returns an event sequence **Seq** such that:

Valid_Seq(P, Events, Seq)

or **None** if no such sequence exists.

6.3.2 Complexity

We claim that the Sequence generation is NP-hard. We will argue this point by informally sketching a reduction of the SAT problem to the Sequence Generation Problem.

Let (V, F) be an arbitrary instance of the SAT problem. Let B be any algorithm that solves the Sequence Generation problem. We thus need to efficiently construct an appropriate function *Events* to serve as input to B, such that B's output gives an indication about the satisfiability of (V, F). Similar to the State Generation reduction, for our construction of Events this indication will be the following:

$$\text{B(Events)} = \begin{cases} Seq & \text{if SAT(V, F) = True} \\ None & \text{if SAT(V, F) = False} \end{cases} \tag{6.2}$$

We construct our Events function in such a fashion that if B succeeds and returns an event sequence Seq, this sequence corresponds to a truth assignment to V that satisfies formula F. Conceptually, this correspondence is of the following nature: Every variable in V is represented by a conductor. Additionally, there is a reference conductor E. Which value Seq assigns to some variable Vj, denoted as Seq(Vj), is indicated by the timing relation the change to conductor E has with respect to Initiate(Vj):

$$\text{Seq(Vj)} = \begin{cases} 1 & \text{if Initiate(Vj)} \leftarrow \text{Initiate(E)} \\ 0 & \text{if Complete(E)} \leftarrow \text{Initiate(Vj)} \\ x & otherwise \end{cases} \tag{6.3}$$

Whereby x indicates that assigning either value to Vj will render the formula F true. Note that the cases Initiate(Vj) $\leftarrow$ Initiate(E) and Complete(E) $\leftarrow$ Initiate(Vj) are indeed disjoint: because of the natural relation Initiate(E) $\leftarrow$ Complete(E), if both cases

were true simultaneously, we would arrive at a cyclic graph (see figure 6.3), which is a contradiction since we have defined event relations to be *strict*.

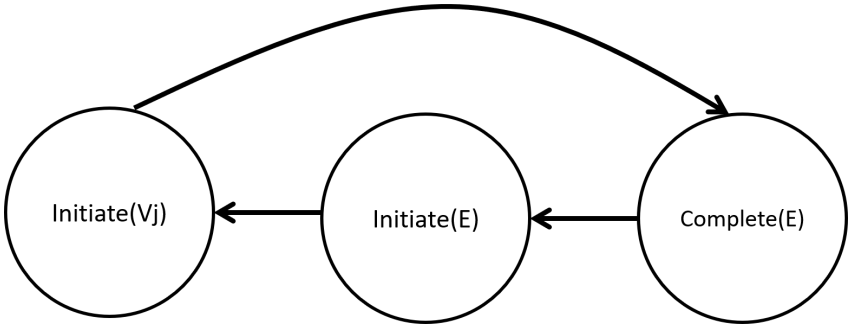


Figure 6.3: The resulting cyclic graph if we happened to be in the first two cases of the definition of Seq(Vj) simultaneously

To visualise this situation, we will represent the change of the conductor C on a time line as follows:

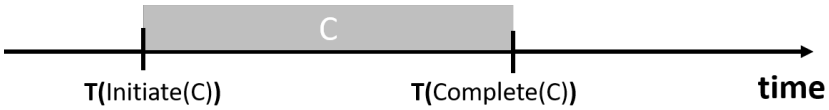


Figure 6.4: Schematic illustration of a conductor change

In the following figure, the colour and pattern of each conductor's change is representative of its implied truth assignment: red and striped for 0, green and dotted for 1 and brown and grid pattern for x. The exception is of course our reference conductor E in blue, which we have shifted slightly down to show the exact timing relations of conductors V4, V5 and E:

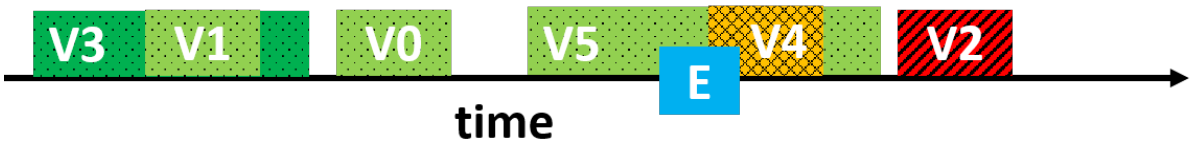


Figure 6.5: An illustration of the correspondence between truth assignments and timing relations to Change(E)

Before we discuss the construction of Events, we wish to make a (trivial) observation about how a truth assignment A can render a clause F[j] in F true. There are two possibilities for A to accomplish this:

1. A assigns value 1 to a variable in positive(F[j])
2. A assigns value 0 to a variable in negative(F[j])

Figure 6.6: The two options that render clause F[j] true

With this in mind, we can now consider the construction of Events: Events is supposed to be a mapping of conductors to Event Graph structures. The conductors we need to represent our SAT(V, F) problem are the following:

- **E**, our reference conductor
- A conductor **Vj** for every variable Vj ∈ V.
- A conductor **F[j]_TRUE** for every clause F[j] in F. The Event Graph of this conductor will enforce possibility (1) as discussed in list 6.6, i.e. it will enforce that for some variable Vi in positive(F[j]), Initiate(Vi) ← Initiate(E).
- A conductor **F[j]_False** for every clause F[j] in F. The Event Graph of this conductor will enforce possibility (2) discussed in list 6.6, i.e. it will enforce that for some variable Vi in negative(F[j]), Complete(E) ← Initiate(Vi).
- A conductor **D[j]** for every clause F[j] in F. With the two conductors F[j]_TRUE and F[j]_FALSE we will be enforcing both possibilities listed in list 6.6, despite one of the two being sufficient to render F[j] true. To remedy this, we add D[j] as a fresh variable and conceptually augment clause F[j] with the two literals D[j] and ¬ D[j].

With this rough idea of the different conductors and their purpose in mind, we define Events as shown in table 6.3. Quite obviously, this construction can be realised in polynomial time of the number of variables n and the number of clauses m.

Note that apart from the natural relation of Initiate(W) ← Complete(W) that holds for every conductor W, only the event graphs of F[i]_True and F[i]_False conductors specify additional event relations. We now wish to establish an informal argument that these indeed accomplish possibilities (1) and (2) (defined in list 6.6) respectively as we claimed above.

Consider the Event Graph of conductor F[j]_True. Initiate(F[j]_True) is defined as an implicit event of type First on the set of events S = (positive(F[j]) ∪ {D[j]}). Let W be the conductor ∈ S whose Initiate event *Initiate(W)* is the first to happen. According to the definition of a *First* implicit event that we have given in section 5.5.6, it follows that Initiate(F[j]_True) = Initiate(W). The event graph of F[j]_True furthermore defines the

Events [Conductors → Event Graph]				
E	→	init	type	Explicit
V0	→	init	type	Explicit
...				
Vn-1	→	init	type	Explicit
D[0]	→	init	type	Explicit
...				
D[m-1]	→	init	type	Explicit
F[0]_True	→	init	type	Implicit
			sub	First
			events	positive(F[0]) ∪ {D[0]}
		aI	{E}	
F[0]_False	→	init	type	Implicit
			sub	Last
			events	negative(F[0]) ∪ {D[0]}
		bI	{E}	
...				
F[m-1]_True	→	init	type	Implicit
			sub	First
			events	positive(F[m-1]) ∪ {D[m-1]}
		aI	{E}	
F[m-1]_False	→	init	type	Implicit
			sub	Last
			events	negative(F[m-1]) ∪ {D[m-1]}
		bI	{E}	

Table 6.3: Events function, any irrelevant attributes have been omitted

after Initiate (aI) attribute to be {E}. According to the definition of said, this enforces the relation Initiate(F[j]_True) = Initiate(W) ← Initiate(E). Since Initiate(W) is the *first* Initiate event to happen, this does not imply anything about the relation of the other Initiate events of conductors in S. Thus, Initiate(W) ← Initiate(E) is equivalent to saying: The Initiate event of at least one conductor in S must happen before Initiate(E), which corresponds exactly to our claim above.

The argumentation for the F[j]_False event proceeds in a very similar fashion, which is why we omit it for the sake of brevity. Since SAT(V, F) is known to be NP-hard, this informal reduction implies that the Sequence Generation mechanism is NP-hard as well.

6.3.3 Approximation

The Sequence Generation problem being NP-hard is not ideal. As we will see in later sections, an NP-hard problem does by no means imply that we cannot solve some instances reasonably efficiently. However, recalling the fact that our Sequence Generation mechanism might fail, we could very well end up calling it on potentially exponentially many solutions our State Generation mechanism is digging up. We will thus not attempt to provide an algorithm that is capable of solving the general Sequence Generation problem. Instead, we propose an approximation of the Sequence Problem in this section, which can be solved in polynomial time.

What makes the general Sequence Generation problem hard is the handling of *Implicit events*, namely deciding which event will be the *first* or *last* to occur. This is precisely what we exploited when reducing SAT to the Sequence Generation problem in the last section. It should be noted that the instance we constructed to solve the SAT problem was a special case itself, since we had no Implicit events referencing other Implicit events.

The approximation we propose avoids these decisions about Implicit Events by stretching our concept of events a bit. Consider an arbitrary implicit event I of the following format:

I [Initiate Event]	
type	**Implicit**
sub	**sub**
events	**E**

Table 6.4: A symbolic Initiate Event instance

Rather than identifying a single command as the definition of Initiate Events suggests (see definition 35), we approximate the Implicit Event I by resolving it to *all* events referenced by E. Time-wise, this approximation of I corresponds to an interval that stretches from T(First(E)) to T(Last(E)). Quite naturally, relations to other events would then be defined as follows:

Let A be an event that is supposed to happen *before* and B an event that is supposed to happen *after* I. Then the following must hold:

$T(A) < T(First(E)) \leq T(Last(E)) < T(B)$

With these relations, we are no longer required to explicitly decide on which event will be the first or last to happen; we only need to ensure that event A happens *before all* events referenced in E and event B happens *after all* events referenced in E. In our graph representation and with E = {E1, ..., En} we can represent this as depicted in figure 6.7.

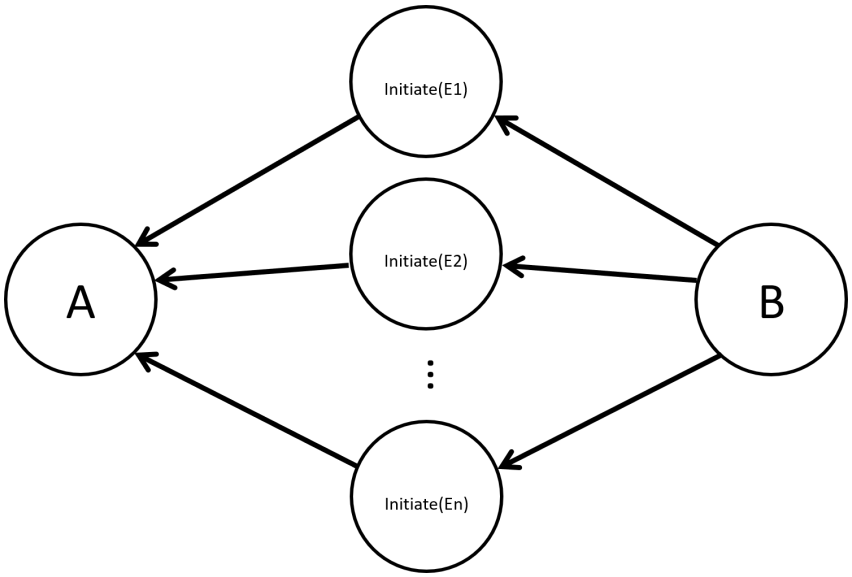


Figure 6.7: Graph representation of our approximation of the Implicit Event I

Considering our original definition of I as either *Last(E)* or *First(E)*, depending on the attribute **sub**, it is obvious that our approximation is sound with respect to both options:

$T(A) < T(First(E)) \leq T(Last(E)) < T(B)$ implies both $T(A) < T(First(E)) < T(B)$ as would be required if sub = First and $T(A) < T(Last(E)) < T(B)$ as required by sub = Last. The soundness of our approximation is important, otherwise we could no longer guarantee the correct construction of event sequences.

We still need to discuss how we can resolve *nested* implicit events. Consider thus the case where E references another implicit event I'. With our approximation being defined as *all* events referenced by E, it is clear that the following equivalence holds, whereby we have denoted the fact that we refer to our approximation of implicit event I by setting the attribute **sub** to "Approximation":

I [Initiate Event]	
type	**Implicit**
sub	**Approximation**
events	**E**

=

I [Initiate Event]	
type	**Implicit**
sub	**Approximation**
events	**(E \ {I'}) ∪ I'.events**

Table 6.5: Two equivalent approximate Initiate Event instances

To argue this more formally, we could make use of our time-based definition and would thus need to show that:
First(E) = First((E \ I') ∪ I'.events) and Last(E) = Last((E \ {I'}) ∪ I'.events)

This is quite easily done, recalling how we defined our **index** function in section 5.9.2 and the fact that for both min and max, it holds that:

$\min(A \cup B) = \min(\{\min(A), \min(B)\})$

For the sake of brevity, we omit this formal argumentation. From now on, we will view all Implicit events as approximations. Based on our claim above, we define a function *Resolve*. Given a conductor W and an *Events* assignment, this function returns the set of events E that Initiate(W) resolves to, whereby E is only allowed to contain already happened or Explicit Initiate events:

Resolve: $\mathit{String} \times (\mathit{P.conductors} \to \mathit{Event\ Graph}) \to \mathbb{P}(\mathit{Events})$

Resolve(W, Events) =

$$\begin{cases} \{\text{Initiate(W)}\} & \text{if Events(W).init = Happened} \\ \{\text{Initiate(W)}\} & \text{if Events(W).init.type = Explicit} \\ \bigcup_{\text{E} \in \text{Events(W).init.events}} \text{Resolve(E, Events)} & \text{otherwise} \end{cases}$$

6.3.4 Algorithm

In this section, we sketch an algorithm that solves the Sequence Generation with approximated Implicit Events as defined in the last section. We split our discussion into three parts: the resolving of Initiate Events, the construction of the global event graph and the generation of the event sequence. Let therefore (**Events**) be an arbitrary instance of the Sequence Generation problem as defined by definition 50.

Resolving Initiate Events Unfortunately, due to potentially cyclic references, we cannot resolve our implicit event in the same recursive manner as the definition of the Resolve function we have given in the last section. Instead, we need to compute a fixed point. In order to ensure that our algorithm runs in polynomial time, we proceed as follows:

We construct an *Initiate Event Graph*. As already implied by the name, the nodes of this graph correspond to Initiate Events: for every conductor W on the platform, there is a node Initiate(W) in our graph. We then insert directed edges as follows:

$\text{Initiate(A)} \to \text{Initiate(B)} \iff \text{Events(A).init.type} = \text{Implicit} \wedge \text{B} \in \text{Events(A).init.events}$

There is the following correspondence between this Initiate Event Graph and our Resolve function: Resolve(W) is equal to all nodes *reachable* from node Initiate(W) that corre-

spond to explicit or already happened events. Thus, we can compute Resolve(W) for every conductor W as follows:

- Compute the transitive closure R of our Initiate Event Graph, which can be done in polynomial time, for instance using the Floyd-Warshall algorithm.
- For all Initiate(W2) such that (Initiate(W), Initiate(W2)) $\in$ R, add Initiate(W2) to Resolve(W) if Events(W2).init = Happened or Events(W2).init.type = Explicit. This can also be done in polynomial time, since the number of edges in our transitive closure is polynomial in the number of conductors.

Constructing the global Event Graph The nodes of our global event graph G correspond to Initiate(W) and Complete(W) for all conductors, with the exception of Initiate(W) for any W whose event graph Events(W) defines an implicit Initiate Event. In other words, we include all *visible* events as well as all already happened events as defined in section 5.9.2.

We add all edges to G that are returned by the function **Edges**, which was also defined in the aforementioned section. However, we modify Edges such that every reference to Initiate(W) is replaced by the set Resolve(W). The resulting definition of Edges is therefore as follows:

Edges: *String* $\times$ *Event Graph* $\to \mathbb{P}$(*Events* $\times$ *Events*)

Edges(W, E) =

$$\begin{cases} \bigcup_{(a,b)\in E.req} \{(\text{Complete(a)}, \text{Initiate(b')}) \mid \text{b'} \in \text{Resolve(b)}\} & \text{E.init = Happened} \\ \Big(\{(\text{Initiate(W')}, \text{Complete(W)}) \mid \text{W'} \in \text{Resolve(W)}\} \cup \\ \quad \bigcup_{bi\in bI} \{(\text{Complete(bI)}, \text{Initiate(W')}) \mid \text{W'} \in \text{Resolve(W)}\} \cup \\ \quad \bigcup_{bc\in bC} \{(\text{Complete(bc)}, \text{Complete(W)})\} \cup & \text{otherwise} \\ \quad \bigcup_{ai\in aI} \{(\text{Initiate(W')}, \text{Initiate(ai')}) \mid \text{W'} \in \text{Resolve(W)}, \text{ai'} \in \text{Resolve(ai)}\} \cup \\ \quad \bigcup_{ac\in aC} \{(\text{Complete(W)}, \text{Initiate(ac')}) \mid \text{ac'} \in \text{Resolve(ac)}\} \Big) \end{cases} \tag{6.4}$$

Remark. Note that we are, in a sense, performing an inverse of the process we devised when defining the validity of a sequence S: If the sequence is provided to us, it suffices to resolve all events to sequence indices and to *then* look at the edges defined by the original event graphs to see if all timing requirements are upheld. Now that we are supposed to construct a valid sequence, we need to correctly generate a global event graph to *then* determine appropriate an appropriate sequence index for every event.

From the above definition, it is clear that the edges obtained by **Edges**(W, E) for all conductors W only reference nodes that exist in our global event graph. Before proceeding to the last step, we need to appropriately handle *already happened* events. Recalling that our index function assigns the smallest value, namely -1, to all already happened events, it follows that any already happened event E must not feature any outgoing edges for the constructed event sequence to be valid. We thus perform the following actions:

For every already happened event E:
    If ∄ event E' such that edge (E, E') is in G:
        Remove node E and all adjacent edges from G
    Else:
        Terminate and return *None*.

Event Sequence Generation We claim that the global event graph G contains all event relations specified by Events. Therefore, we can obtain a correct event sequence S as follows:

If G is acyclic:
    S = topological_sort(G)
    return reverse(S)
Else:
    return *None*

Note that due to our removal of all already happened events from G, the returned sequence reverse(S) indeed observes Format(Events, S), which corresponds to the first condition of Valid(Events, S). We do not provide a proof for the second condition.

6.4 Consumer Demand Generation

6.4.1 Problem Definition

As always, we recall the informal problem definition of the problem that we have given in section 4.2.1:

> Generate a feasible interleaving of all consumer transitions that are requested simultaneously.

We have already discussed the correctness of such an interleaving in section 5.9.3. We thus merely need to discuss what *feasible* is supposed to mean in this context. This is quite straightforward: As we have seen in the last sections, problem instances exist for both the State and Sequence Generation mechanism that result in *None* being returned since no *valid* solutions exist. Feasible thus means all the problem instances generated by the interleaving have *valid* solutions.

We thus arrive at the following definition, whereby as in definition 46, every element (C, P1, P2) in **Trans** denotes that consumer C should transition from its power state P1 to P2:

Definition 51 (Consumer Demand Generation problem)**.** Given a Platform P and a set of transition descriptions $Trans \in \mathbb{P}(\text{String} \times \text{String} \times \text{String})$, the Consumer Demand Generation mechanism either returns a **feasible** sequence of consumer demands S such that

Valid_interleaving(Trans, S)

Or returns *None* if no such sequence S exists.

6.4.2 Complexity

The question of complexity is quickly answered for the Consumer Demand Generation Problem; the *feasibility* of an interleaving depends on the feasibility of the resulting State and Sequence Generation problem instances. Recalling that all our reductions to SAT were exactly based on said feasibility question (since SAT, in contrast to State and Sequence Generation is a pure decision problem), it is obvious that Consumer Demand Generation must be NP-hard too.

6.4.3 Algorithm

In this section, we sketch an algorithm for the Consumer Demand Generation problem. We put the focus of this sketch on the exploration of the different interleavings. Therefore, we will abstract determining the *feasibility* of an individual sequence step with a predicate **Feasible**, since that just involves making the appropriate calls to State and Sequence Generation mechanisms and checking the result for None.

Let thus (P, Trans) be an arbitrary Consumer Demand Generation problem. In order to efficiently perform the aforementioned exploration of interleavings, we use an approach based on dynamic programming (DP). We construct the usual DP-table **DP** as follows:

Our table will feature a dimension for every transition in **Trans**. Let (C(i), P1(i), P2(i)) be the i-the element of Trans. The i-th dimension of our table will have the following character:

Let T(i) be the sequence of consumer demands defined by C(i).trans(P1(i), P2(i)). We define an *augmented* sequence T'(i) to be

T'(i) = C(i).P1(i) + T(i) + C(i).P2(i),

meaning that we add the consumer demands for the initial power state P1(i) to the beginning and for P2(i) to the end of T(i). The i-th dimension will feature a *row* for every element in T'(i).

For Trans = {(CPU, Off, On), (FPGA, Off, On)} and with the abbreviations CPU.trans(Off, On) = {CPU1, ..., CPUn} and FPGA.trans(Off, On) = {FPGA1, ..., FPGAm}, our table might look as follows:

	CPU.Off	CPU1	...	CPUn	CPU.On
FPGA.Off					
FPGA1					
...					
FPGAm					
FPGA.On					

Table 6.6: An example DP table

Conceptually, the element at index (i(0), ..., i(len(Trans) -1)) of our table thus represents the union of consumer demands associated with the corresponding *rows*:

$$\bigcup_{j\,=\,0}^{\text{len(Trans) - 1}} \text{T'(j)[i(j)]}$$

Thus, DP[0, ..., 0] corresponds to a consumer demand that enforces a platform state **before** the transitions in Trans are executed and where every consumer is still in its initial state P1. Similarly, the last element of DP enforces a platform state **after** Trans where every consumer has attained power state P2.

In the context of table DP, Consumer Demand Generation boils down to finding a valid and feasible path through the table from index (0, ..., 0) to the last element at index (len(T'(0)) - 1, ..., len(T'(len(Trans) - 1) - 1)). A *valid* path is thereby defined as follows:

Definition 52 (Valid Path)**.** Let P be a sequence of elements in DP at indices I(0), ..., I(m). We consider P to be valid if:

$\forall$ j $\in$ {0, ..., m-1}**.** I(j+1) - I(j) $\in (\{0,1\}^{len(Trans)} \setminus \{0\}^{len(Trans)})$

In other words, with every step we take through our table, the index of dimension d must either increase by one or remain the same. Of all dimensions, the index of at least one should increase (otherwise we would remain at the same index).

Note that there is an exact correspondence between a valid path through our table and the Valid_trans predicate we defined in definition 45: The indices i whose existence the predicate demands can be extracted from a valid path as follows: Consider the j-th transition in Trans and a valid path through the table with indices I(0), ..., I(m). Then, index i(x) as defined in definition 45 can be computed as follows:

Since our valid path starts at (0, .., 0) and ends at the *last* element of DP, it follows that I(0)[j] = 0 and I(0)[j] = len(T'(j)) - 1. By definition of a valid path, the indices of dimension j must either increase by one or remain the same in every step of the sequence I(0)[j], ..., I(m)[j]. Thus, there must especially exist a y $\in$ {0, ..., m - 1} such that I(y)[j]

= x and I(y+1)[j] = x + 1. According to our construction of T', the x-th transition step of trans(C(j), P1(j), P2(j)) corresponds T'[x+1], and thus i(x) corresponds to index (y+1) of the valid path.

Using our DP-table, we construct such a valid path incrementally, by storing in each table entry information on whether we have already discovered a prefix of a valid path that ends at the corresponding entry. We traverse the table in a multi-dimensional equivalent of row or column-major order and when considering the entry at some index I, we update our table as follows:

If DB[I] indicates that a prefix of a valid path ending there was found:
    For every $\delta \in (\{0,1\}^{len(Trans)} \setminus \{0\}^{len(Trans)})$:
        If Feasible(Step from DB[I] to DB[I + δ]):
            Update DB[I + δ] accordingly

If the last entry of DP indicates that we have found a valid path ending there, we find the corresponding path by backtracking through the table using the information stored in every entry and return the corresponding consumer demand sequence. If not, we return None.

Note that due to the Sequence Generation Mechanism depending on the initial platform state (see the construction of *Events* in section 50), a general Consumer Demand Generation mechanism will need to keep track of the precise Platform State we reached in every table entry. See section 8 for description of how our implementation circumvents this.

The described DP mechanism can thus find a valid and feasible interleaving in time proportional to the dimensions of table DP, the state stored in every entry of DP and the time it takes to evaluate the Feasible predicate.

6.5 Summary

In this chapter, we have discussed the three mechanisms that need to be built on top of our platform model to generate correct static management actions. We have realised that all of the general problems our mechanisms are required to solve are NP-hard. In the case of State and Consumer Demand Generation, we have presented an algorithm that solves this general problem. For Sequence Generation, we have simplified our problem formulation to yield an approximation of the general solution that can be found in polynomial-time.

Chapter 7

Modelling the Enzian

7.1 Overview

As already mentioned in section 2.2, the Enzian platform features a server-class ThunderX CPU and as well as a high-end FPGA. As one would expect, the platform supporting these agents is not entirely trivial. Figure 7.1 depicts the components and conductors of the Enzian platform as drawn by the Graphviz graph visualisation software. For the sake of simplicity, purely monitoring-based connections are not shown.

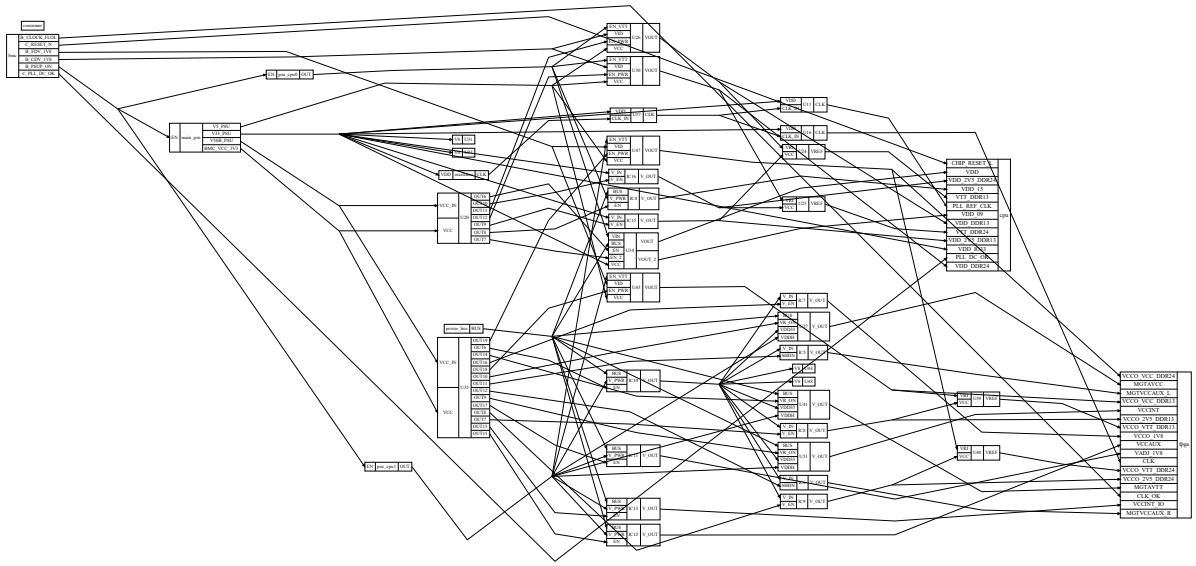


Figure 7.1: An automatically generated picture of the Enzian platform's components and conductors.

In the next two sections, we will discuss why we consider the Enzian platform to be well-designed and give some general advice on how model instances should ideally be created.

7.2 Why the Enzian platform is well-designed

The platform of the Enzian is designed in a manner that avoids almost all of the concerns we have voiced over the last few chapters.

7.2.1 Monitoring Functionality

The Enzian platform offers a wealth of monitoring functionality. Most notably, it includes two ispPAC-POWR1220AT8 manufactured by Lattice Semiconductor which feature twelve VMON inputs that can be used to measure the voltage across connected conductors [23]. Both these ISPPACs are connected to the always-on power rails of the main power supply and a dedicated I^2C bus and thus their monitoring functionality is always available.

This always-on monitoring functionality is ideal for the implementation of Complete events. If we had to rely on functionality like READ_VIN as defined by the MAX15301, more in-depth modelling of such monitoring actions would be required: READ_VIN is a PMBus command and consequently we would need to ensure that the bus is available and the MAX15301 powered to make use of it. Ultimately, this would require us to introduce *soft* event restrictions to ensure that, whenever possible, at least one monitoring functionality is available to confirm a complete event.

7.2.2 I^2C Buses

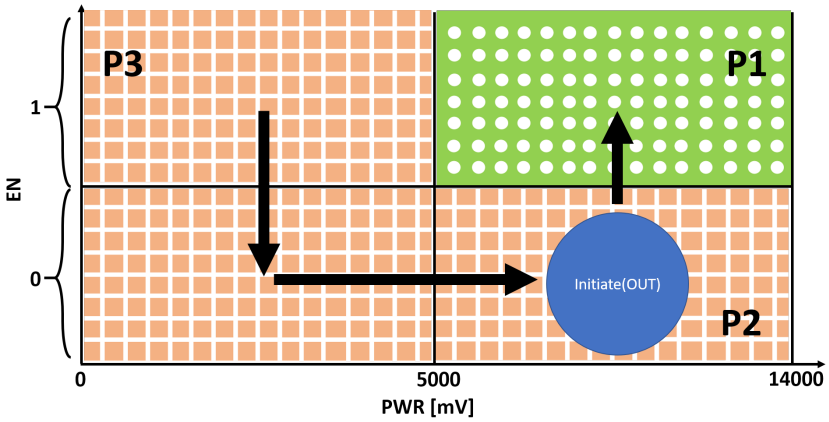
The Enzian features several communications buses. The producers connected to each of these have been chosen in a manner that no powered-down device on the bus ever thwarts communication with another powered device on the same bus.

On the Enzian, buses are hence available in every situation where we might desire to make use of them. This avoids all the concerns we had in connection with the modelling of buses in section 5.6.

7.2.3 Enable Signals

The Enzian's platform generously provides a dedicated enable signal for *every* output produced by a producer. This allows us to elegantly solve another problem we faced, this time in connection with events and ideal static management.

Let us recall the situation associated with the following figure:



In section 5.5.7, we discussed how we might construct sequence requirements for the MAX15301's OUT pin that adhere to ideal static management (see definition 34). Unless we are fine with OUT adopting its current default state (transiently, at least), there is no way to transition from P3 to P1 in an *valid* fashion. Thus, when entering State Possibility P3, we automatically restrict the next stable state OUT will adopt (transiently) to be either 0V or the default state.

The only situation where we might need to transition to state like P3 happens on a platform where two or more producers have a shared enable signal but individual power supplies. On the Enzian, with its dedicated enable signals, this clearly does not happen, so we avoid entering states like P3 by not even modelling them in the first place. This has the following, additional advantage:

Consider the level plot of the MAX15301. Without P3, the *red* region that symbolises that the voltage of OUT is 0V takes on a convex shape. This has two major advantages: Firstly, we could now model both the *green (dotted)* and *red (grid pattern)* region with a single State Possibility, which favourably improves the runtime of our State Generation algorithm. Secondly, this convexity prevents the need for additional requirements to be imposed on *already happened* events if we wish to adhere to ideal static management.

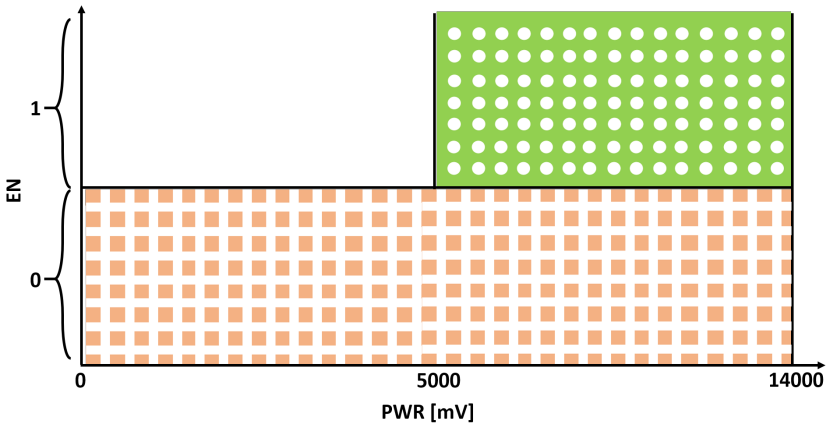


Figure 7.2: Simplification of the MAX15301's State Possibilities possible on the Enzian

7.2.4 Hierarchy

The platform of the Enzian is strictly hierarchical. This can be seen quite easily by studying the visualisation of the Enzian's topology in figure 7.1: If we disregard purely monitoring-based connections such as said to the VMON inputs of the ispPAC as mentioned in section 7.2.1, we find that the multi-graph obtained by taking components as nodes and conductors as directed edges is acyclic.

This is not such a huge surprise; a cyclic dependency in a power and clock management platform most likely hints at a certain degree of self-management of the involved components, which in turn implies that the platform should be modelled differently.

But regardless of whether a strict hierarchy is to be expected or not, the fact that Enzian happens to be one lets us resolve implicit events a bit more efficiently. We discuss this in more detail in section 8.2.2.

7.3 Dependence of CPU and FPGA

Studying the visualisation of the Enzian platform, it looks like CPU and FPGA could be powered independently of each other: both seem to sit behind an individual wall of producers dedicated to them. This is, however, not completely true: The 3v3_psup rail of the main power supply directly supplies the VDD_IO input of the ThunderX, while also serving as supply for some of the FPGA's producers. Therefore, the CPU can be booted individually without issue (we simply do not enable the producers on the FPGA side) but if the same is possible for the FPGA depends on the *robustness* of the CPU's power sequence.

Powering VDD_IO along with enabling the main clock signal consists of the first step in the ThunderX's boot sequence [7]. To determine to which degree it is possible to boot the FPGA independently, the following questions need to be answered:

- Deferring the bootstrapping of the CPU: Can the ThunderX boot correctly even if an arbitrarily long delay is inserted between steps one and two of its power sequence?
- Only booting the FPGA: Is it safe to bring VDD_IO online if we do not intend to boot the ThunderX and thus do not complete its boot sequence?

Depending on the answers, our consumer demands must be formulated differently, either allowing or disallowing the FPGA to boot independently, possibly with the introduction of a new *Ready* CPU power state that ensures that step 1 of the CPU's boot sequence is executed correctly such that the CPU can later on be brought up without power cycling the FPGA.

Our model currently enforces the most conservative option that disallows the FPGA from being booted independently.

7.4 The Modelling process

Using the example of the Enzian platform, this section discusses some of the modelling pitfalls as well as what constitutes a *good* platform model.

7.4.1 Consumer Transitions

It is crucial that descriptions of consumer transitions are adapted to the capabilities of the platform. As an example, we take a closer look at the first step in the ThunderX's power sequence that was already mentioned above. The manual instructs that the main clock should be enabled before or while VDD_IO is brought online [7].

As we have seen in our discussion about events, it is difficult if not impossible to enforce that two conductor changes happen *at the same time*. 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.

7.4.2 Sequence Generation

Recall the observation we made in context with producers about the *unpredictability* of the next output they might need to provide (see observation 13). Ideally, no matter what kind of consumer demands C are imposed next, static management should be able to realise them as long as it is within the capabilities of the platform. In particular, this means that the following should hold true:

Statement 53. Let (P, C) be an instance of the State Generation problem. If we task our Sequence Generation mechanism with returning all solutions for instance (P, C) and it finds a non-empty set S of solutions, our Sequence Generation mechanism should be able to generate an event sequence for at least one solution in S.

Remark. Note that it does not make sense to ask that our Sequence Generation mechanism return an event sequence for *every solution* in S: Since most of our platform is transparent to consumers (see observation 15), most of the platform state that we generate is transparent to them too and therefore not every valid platform state must truly be attainable to fulfil any sensible consumer demands.

Whether or not we adhere to the above statement depends on the quality and strictness of the sequence requirements provided by the producer descriptions. Therefore, we arrive at the following modelling advice:

Advice 54. Vary the strictness of sequence requirements until an adequate compromise between adhering to ideal static management and the above statement is found.

Due to the Enzian platform being very well-designed, we claim that our Enzian platform instance achieves a bit more than the above mentioned compromise:

Claim 55. *We claim that the sequence requirements described by our platform instance of the Enzian have the following properties:*

- *They adhere to ideal static management.*
- *For every solution found by our State Generation mechanism, a valid event sequence is found.*

Chapter 8

Implementation

This chapter is dedicated to the implementation we constructed as a proof of concept. Our mechanisms have been written in Python and, for the sake of convenience, expect model instances that are also specified in Python. In the next few sections, we discuss different implementation aspects in more detail.

8.1 Model syntax

We leverage Python's object oriented features for the descriptions of components: Every type of consumer and producer is described using the static attributes of a dedicated python class. A platform instance can then be created by instantiating the appropriate descriptions and by specifying the conductor connections between component instances.

For the sake of better readability and easier specification, the model syntax our implementation expects deviates slightly from the one presented in the modelling chapter. The remaining subsections focus on these differences:

8.1.1 Conductor States

Our implementation makes use of the same simplifications as we have introduced in section 5.3: direct currents are specified using a single voltage dimension and logical signals using the set {0, 1} rather than the general *high, low and freq* expression with which we have defined Conductor States. To ensure that these simplifications are understood correctly, our implementation requires each input and output to specify its *state type* as *logical, dc, clock, or ac*.

We have introduced some abbreviations that allow for a simpler (and more storage efficient) representation of State Spaces: For two integers a $\leq$ b, the set {a, a+1, ..., b-1, b} containing all integers between a and b (inclusively) is abbreviated as a tuple (a, b). Note that in order to avoid confusion between such a range and the representation of a State Space or Conductor State, we require the latter to be specified as (nested) *Python lists*.

For the sake of easier interpretation by our implementation, we require every State Space

dimension to be either specified as a set or an aforementioned range. To mitigate this more cumbersome notation, we interpret these sets in a *Cartesian product* fashion for multi-dimensional State Spaces (such as clock signals). Thus, the notation

$$[[\{0,1\},(4,6)]]$$

is equivalent to:

$$[[\{0\},\{4\}],[\{0\},\{5\}],[\{0\},\{6\}],[\{1\},\{4\}],[\{1\},\{5\}],[\{1\},\{6\}]]$$

8.1.2 State Requirements

Rather than imitating the State Requirement structure proposed by our model, our implementation makes use of Python dictionaries whereby the key corresponds to the *id* and the associated value to the *state* attribute of a State Requirement structure.

8.1.3 Complex constraints

Occasionally, we need to be able to express certain relations that must be observed by inputs and outputs of a certain producer. An example for such a producer present on the Enzian is the NCP51400 [24]. Based on an input voltage VDD, it outputs the corresponding reference (VREF) and termination (VTT) voltages for DDR SDRAM.

According to the description of DDR SDRAM voltage termination found in [12], the following relations exist between VDD, VREF and VTT:

$\text{VDD} = \text{VREF} = \frac{1}{2}\ \text{VTT}$

Of course, our producer description can accommodate arbitrary such constraints by hard-coding all possible state combinations into individual State Possibilities. In the case of the NCP51400, this would result in up to 2600 such Possibilities, since it allows for VDD to be in a range from 1V to 3.6V. Considering the algorithm we proposed to solve the State Generation problem in section 6.2.3, which exhaustively tries all such Possibilities, it is clear why this representation is not such a good idea in practise.

For this reason, our implementation allows for the specification of arbitrary additional constraints, which will be passed to the Z3 constraint solver [1] along with the *State Space* mapping S returned by the procedure described in 6.2.3.

8.1.4 Default States

As mentioned in section 5.6.3, our implementation is responsible for the correct inclusion of default states. For this purpose, our implementation allows every State Possibility P to specify a *state_update* function. After every static management action, P.state is updated with the State Space returned by that function.

8.1.5 Representation of Events

The representation of sequence requirements has been simplified in accordance with the nature of the Enzian topology: Since according to claim 55, our description adheres to ideal static management and because of the convexity of equi-output spaces discussed in section 7.2.3, we can make the following observations about events on the Enzian:

- If a conductor's state does not change, its Complete and Initiate events have automatically already happened and no further requirements are needed.
- If the state of a conductor C1 changes, but stays within the State Requirements for C1 that the chosen State Possibility P for conductor C2 specifies for C1, then C1's change has already happened from the perspective of C2. This means that if the Event Graph of P requires something to happen strictly after the change of C2, Sequence Generation would fail.

These two observations capture the majority of the Initial State dependency that most sequence requirement functions would need to specify. Our model implementation thus requires State Possibilities to specify a single Event Graph. In the few cases where additional initial state dependencies need to be specified, a *dependency_update* function can be provided, analogous to the state_update function described in the previous section.

8.1.6 Consumer Transitions

Unlike the interpretation we have used in our model and mechanism discussions, our implementation considers specifications of consumer transitions to be *incremental*. Thus, a conductor W need only be mentioned in a transition step S if the consumer demands concerning W change in transition step S.

8.2 Mechanism Implementation

This section discusses how we have implemented the mechanisms detailed in chapter 6.

8.2.1 State Generation

We have implemented the State Generation algorithm precisely as presented in section 6.2.3. In the following subsections, we will comment on some runtime improvements we have developed for said algorithm, as well as how we have cast the State Generation to a z3-solver problem instance.

Advanced Backtracking The stack-based backtracking mechanism the procedure described in 6.2.3 hints at can be extremely inefficient. Consider the following situation: Our platform features n conductors W(1) to W(n). We consider the following, fictional execution state of our State Generation procedure for this platform:

Desired_States = {req} with req.id = W(n) and req.state = S. Unfortunately, conductor W(n) does not possess any State Possibilities that agree with state S, so our procedure

will inevitably fail in the next step. This unfortunate desired state was imposed on W(n) by the currently chosen State Possibility for **W(1)**, Poss(W(1)). Now, consider the following Choices stack:

Choices

Associated Conductor	Execution State	Remaining Possibilities
W(n-1)	E(n-1)	1
		4
		16
W(n-2)	E(n-2)	3
		4
		5
		8
...		
W(1)	E(1)	1

Stack bottom

Table 8.1: An example stack of execution states

Whereby the *associated conductor* references the conductor for which we have pushed the *remaining state possibilities* in Options along with the *current execution* state onto Choices on line 5 (see section 6.2.3). The number in the column *Remaining Possibilities* defines the index of the corresponding State Possibility. For instance, the 1 in the first row indicates that the *first* State Possibility of W(n-1), i.e. W(n-1).poss[1], constitutes another State Possibility worth exploring.

If we simply *pop* the next Possibility to try from our stack as described by our procedure, we need to work through the entire Choices stack before our procedure reconsiders the choice of Poss(W(1)) that causes it to fail because of the state S imposed on W(n). This *working through the stack* is by no means a linear process: For every of the 4 remaining possibilities for W(n-2), we will *discover* new Options for conductor W(n-1) that will be added to our stack and will consequently need to be tried first.

Considering this example, it is obvious that with naive stack-based backtracking, the performance of our State Generation procedure heavily depends on the sequence in which choices about State Possibilities are made, i.e. the sequence in which we attempt to enforce our requirements in *Desired_States*.

We can improve this situation by including some *contextual information* into our Desired_States. Whenever we add an additional State Requirement to Desired_States, we could for instance include the conductor whose choice of State Possibility was the cause of this requirement. In the example above, we would therefore note that *req* was caused by conductor W(1). Consequently, we could directly discard the whole stack up to the entry for W(1), since all State Possibility decisions and computations made since execution state E(1) might have been influenced by the impossible choice Poss(W(1)) and are therefore worthless.

In practise, the correct implementation of such an *advanced backtracking* mechanism is not entirely trivial: we need to be very careful with discarding portions of our stack, especially if we require our procedure to return *all* possible solutions.

Likelihood Heuristic A very simple heuristic that we can integrate into our State Generation procedure is to always consider different State Possibilities *in order*. In other words, of all State Possibilities present in Options as constructed in line 2, we always try the option with the smallest index first. This of course requires our producer descriptions to order State Possibilities accordingly:

Advice 56. Order each Output's State Possibilities according to decreasing likelihood.

Integration of z3 Next to our State Generation procedure as described in section 6.2.3, we have implemented an alternative mechanism. This alternative mechanism consists of a *transformation* of the State Generation problem instance to a global constraint problem that is then handed to the Z3 SMT solver [1]. In this section, we very roughly sketch the nature of this translation.

The main ingredient of a Z3 problem formulation are atomic variables that are combined to formulas using symbols defined by theories.[5]

For such a problem formulation to solve the State Generation problem, we need to be able to extract the resulting Platform State S and the State Possibility assignment Poss from the variable assignments returned by the solver. For every platform conductor W, we thus introduce the following variables:

- A variable Poss(W) for the state possibility chosen for W
- Three variables High(W), Low(W) and Freq(W) to represent the three attributes of a Conductor State structure

Using these variables, we can then precisely implement the conditions of the State Generation problem using the standard logic and arithmetic theories provided by the Z3 solver.

Comparison of mechanisms Our implemented mechanisms feature different strengths and weaknesses. We will discuss these based on the different *purposes* for which we will make use of them in the context of our other mechanisms:

- **Finding *all* solutions** For this purpose our own State Generation procedure is better suited. As discussed in our mechanism section, we can very easily and naturally extend it to return all possible solutions. The Z3 solver on the other hand offers no explicit support for the extraction of all solutions. Instead, we would need to impose more and more additional constraints to prevent the solver from returning already seen solutions.

- **Deciding if a solution exists**. Our own State Generation procedure is not very well suited to this problem. Despite advanced backtracking, to truly confirm that no solution exists our procedure will need to actively eliminate every possible combination of State Possibilities. This, depending on how quickly conflicts manifest themselves, might take a considerable amount of time. Z3 directly supports the checking of the feasibility of a problem. Of course, if this check is more performant heavily depends on its concrete implementation, but the Z3 solver is heavily optimised and makes use of sophisticated heuristics.

- **Finding a *good* solution** Occasionally, we might perceive some Stable Platform state to be *better* than another. For instance when it comes to recommended operating conditions, it is best if we choose a voltage that is a far away from the over- and under-voltage lockout values as possible, i.e. ideally some mean value. Similarly, we ideally pick a regulator's default voltage whenever possible, since we thus need to issue fewer PMBus commands and can transition to the new stable platform state more quickly. While the Z3 solver allows for the specification of so-called soft-constraints to express such preferences, we believe that our own State Generation procedure offers more natural control over the solutions found. Firstly, it returns a State Space assignment from which we can pick any state that suits our needs best. Secondly, instead of making use of the likelihood heuristic explained above, we can order the different State Possibilities to reflect our solution preferences.

8.2.2 Sequence Generation

In this section, we will discuss our implementation of the Sequence Generation mechanism as well as its integration with the State Generation mechanisms discussed in the last section.

Implementation The implementation of the Sequence Generation mechanisms *mostly* corresponds to the procedure sketched in section 6.3.4. The main difference is the resolution of Implicit States. Generally, an Implicit Event defined by an output of a producer P mostly references events concerning *inputs* of producer P. This is particularly true for the producers on the Enzian. Furthermore, as remarked upon in section 7.2.4, the Enzian's platform topology is strictly hierarchical, the resulting *Implicit Event graph* is guaranteed to be acyclic. The resolution of Implicit events can thus be performed a bit more efficiently by resolving the different events in a reversed topological order.

Integration with State Generation Recalling modelling advice 54, the description of sequence requirements should ideally be performed in a manner that adheres to statement 53, i.e. that for at least one solution returned by the State Generation mechanism, the Sequence Generation mechanism successfully returns a valid command sequence. If we assume that this is the case, this immediately implies that we must tie the two mechanisms together as follows:

- If our own **State Generation procedure** is used, State and Sequence generation are tied together as described above. Consequently, we are guaranteed to find a valid command sequence if our platform description adheres to statement 53. This is the **recommended** choice.
- If the **Z3-based mechanism** is picked, our implementation will only pass the initial solution returned by the Z3-solver to the Sequence Generation mechanism. No attempt to extract any further solutions from the Z3-solver will be made, and hence the success of Sequence Generation cannot be guaranteed even if our model adheres to statement 53.

Figure 8.1: The consequences the choice of State Generation mechanism has

We continue to request a *new* solution S of our State Generation mechanism until Sequence generation succeeds on S.

Thinking back to the comparison of the two State Generation mechanisms at our disposal, our own State Generation mechanism is well-suited for this purpose, since providing as many alternate solutions as desired is a natural extension of it.

However, for the sake of flexibility, we leave the choice of which mechanism is supposed to be used to the user. We will discuss how this choice is communicated to the implementation in section 8.2.4. The following consequences are associated with the different options:

8.2.3 Consumer Demand Generation

Our implementation is a more efficient but also more restricted version of the Consumer Demand Generation algorithm sketched in section 6.4.3. We justify this restriction based on the modelling advice 54: If we assume that that our sequence descriptions adhere to statement 53, our Consumer Demand Generation mechanism need not worry about the success of Sequence Generation.

As already mentioned in section 6.4.3, this assumption prevents us from needing to keep track of the potentially exponentially many platform states with which we reached a certain DP-table entry. Furthermore, the **Feasible** predicate is reduced to a call to the State Generation mechanism to see if a valid solution exists. The implementation of this Feasible predicate is thus an ideal use case of our Z3-based State Generation mechanism.

8.2.4 Implementation boundaries

Since our implementation has not been fully integrated into the Enzian's management software, this section describes its *implementation boundaries* and the resulting usage. We

split this discussion into two parts, which are describing the input and output boundaries respectively.

Output boundary The format of bus commands expected by different producers is very specific and fairly complex. Similarly, the implementation of bus protocols and the correct handling of gpio interfaces is non-trivial. Therefore, our implementation merely generates a sequence of python commands that leverage the driver functionalities implemented by the existing power and clock management solution on the Enzian.

Input boundary Our mechanisms are not capable of directly interpreting any standard commands defined by the management interfaces discussed in section 3.4.1. Instead, our implementation exposes the following methods that can be called on a platform instance.

Note that since our mechanisms do not directly update the state of the real physical platform, the platform instance is keeping track of the platform state as it *would* be if the generated management actions had been applied. To avoid confusion, we refer to this as the *virtual* platform state.

- **parametrised_state_search**(Demands, Flags) Executes the tied-together State Generation and Sequence Generation mechanisms (see section 8.2.2) on the given **Demands**, the current virtual platform state and as parametrised by **Flags** and returns the management actions found without updating the virtual platform state.
- **apply_changes**(Demands, Flags) Wrapper for parametrised_state search that additionally updates the virtual platform state
- **stateful_node_update**(Transitions, Flags) Performs Consumer Demand Generation for the requested transitions and proceeds to call apply_changes on every step of the generated interleaving.

The following table summarizes the user-available **Flags** that can be passed to the above methods.

Flag	Default	Meaning
use_z3	False	if the Z3-based mechanism should be used, with consequences as detailed in figure 8.1.
all_solutions	False	if *all* valid command sequences should be returned or not. If passed to apply_changes or stateful_node_update, the command sequence that makes the fewest changes to the current platform state is chosen (for every transition step). Ignored if use_z3 is set.
extend	True	Adds trivial state requirements for all conductors and consequently ensures that the consumer demands follow assumption 49 (see section 6.2.3)
advanced backtracking	True	If advanced backtracking should be used. Ignored if use_z3 is set.

Table 8.2: The set of flags that can be passed to the mechanisms

Chapter 9

Evaluation

In this chapter, we attempt to evaluate the feasibility of the static management approach we have presented in the previous chapters. Unfortunately, this is not such an easy task: the detailed design schematics required to properly model another platform are generally not made publicly available. Thus, our only other point of reference is the Enzian platform and its existing implementation of power and clock management.

9.1 Performance of our Mechanisms

This section is dedicated to the performance of our mechanisms. To our knowledge, there is no other approach to static management beyond hard-coded point solutions. Consequently, we lack a point of reference to compare our mechanism performance to, which will inevitably result in this particular evaluation being detached from the current state-of-the-art.

Without an external point of reference, the exact runtime of our mechanisms is not necessarily relevant: If we intend to use our approach in a purely *offline* manner to pre-generate command sequences for different management scenarios that are consequently hard-coded in the BMC firmware, it would not be a huge tragedy if we were forced to wait a long but *manageable* time for the results. The situation is completely different if we were interested in using our mechanisms to perform *online* static management. In that case, performance is critical and we would need to establish and prove hard runtime bounds.

According to this distinction, we will structure our runtime measurements into two categories: One performed from an *online* static management angle discussed in the next subsection and the other to reason about whether we achieve a *manageable* runtime required for an offline application.

9.1.1 Online Static Management

This section is dedicated to evaluating our approach from an *online* management perspective. As already hinted at, this evaluation will not attempt to achieve the rigour and

strictness that is truly required for any real-time application. Instead, we design our runtime measurements in a manner that they provide some very *empirical and rudimentary* insight into whether we *could* be successful in establishing hard runtime bounds for our static management implementation on the Enzian platform *specifically*.

For this empirical evaluation, we will not be concerned with the performance of Consumer Demand Generation. Consumer transitions are fixed management actions and can therefore, even in the case of an online static management approach, be pre-computed offline. We will instead focus on executions of the tied-together State and Sequence Generation mechanisms (as described in section 8.2.1). To arrive at our intended experimental design, we must firstly make some general observations about the runtime of these mechanisms:

First of all, we recall that our Sequence Generation mechanism solves an approximation of the true NP-hard problem in polynomial time. For the following performance evaluation, we will thus assume that the runtime of said is not such a huge concern and behaves more or less predictably. Therefore, we can focus our efforts entirely on bounding the runtime of the State Generation mechanism, which is solving the general NP-hard problem.

Furthermore, we recall claim 55. There, we stated that our sequence requirements for the Enzian platform could be designed in an ideal manner, i.e. such that we adhere to ideal static management and for every solution of our State Generation mechanism, a valid command sequence can be found. This immediately implies that for arbitrary consumer demands, only a *single* State Generation solution needs to be found and passed to Sequence Generation. Together with the above argumentation, this means in particular that we need not worry about the effects different initial platform states have on the measured runtime.

With initial platform states being irrelevant, the only problem that keeps us from establishing a meaningful empirical runtime bound is the wealth of different consumer demands an *online* static management actions would potentially need to deal with. Since the implementation of the Z3-solver is a black box from our point of view, we cannot make any statements about the effect these different demands might have on its performance. However, for our own State Generation procedure, we can make the following observation:

Observation 57. Let C1 and C2 be two sets of consumer demands such that C2 $\subseteq$ C1 (with $\subseteq$ as defined in section 5.9.3). Then, the worst case runtime our State Generation procedure exhibits when tasked with finding *all* solutions for demands C1 is an upper bound for the worst case runtime it requires to find one solution for demands C2.

That the above observation holds is fairly obvious: Let S1 be the set of solutions for demands C1 and S2 the set of solutions for demands C2. Then S2 $\subseteq$ S1, and therefore the worst case runtime required to retrieve set S1 will naturally be an upper bound for the time it takes to retrieve a single element of S2.

Note 58. One might be tempted to think that the worst case runtime required to retrieve a single solution for C1 already provides an upper bound for finding a single solution for C2. More concrete consumer demands restrict the space of State Possibilities that must be searched, and it seems intuitive that this should also speed-up the process of finding one solution. This is, however, not generally true: Consider the following toy example of a platform consisting of two conductors W and W': In figure 9.1, we have marked valid State Possibility combinations of W and W' with a green dot, and the Possibilities allowed by consumer demands C2 and C1 with an orange and blue frame respectively.

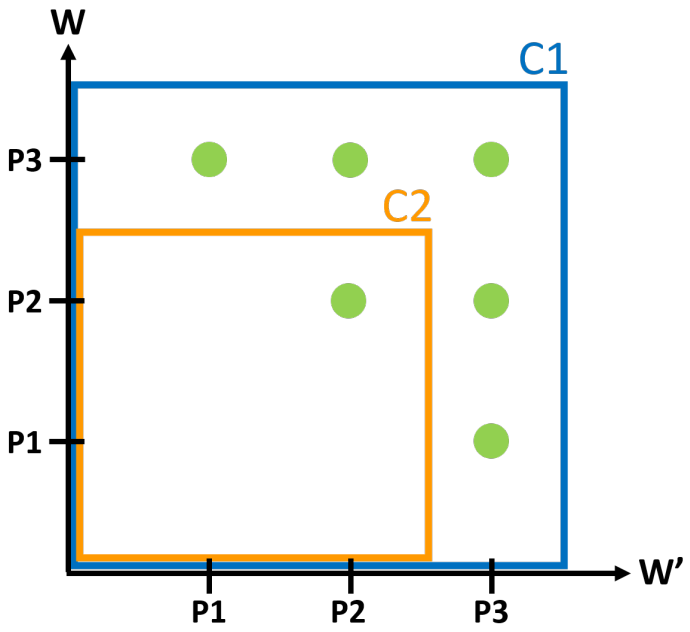


Figure 9.1: A plot visualising the fact that a smaller search space does not necessarily imply that a solution will be found more quickly.

As this plot shows, for consumer demands C1, our State Generation procedure will never need to revert: No matter which possibility we choose for W or W', the combination with P3 of the other conductor will yield a valid solution. The situation is different for consumer demands C2; if we happen to choose P1 for either conductor, we will inevitably be required to revert. Consequently, the worst case runtime for finding a single solution for demands C1 does not provide an upper bound for the worst-case time required to find a solution for C2.

Thus, to obtain an estimate on the worst-case performance of our State Generation procedure for any problem online static management must solve, it suffices to consider the worst-case performance of finding *all* solutions for the most general problems. These problems are statically known: they correspond to all combinations of power states the consumers of a platform can assume. On the Enzian, these would be the following problems:

Problem designation	CPU power state	FPGA power state
P1	Powered on	Powered on
P2	Powered on	Powered off
P3	Powered off	Powered off

Table 9.1: An overview of problem instances P1 to P3

Remark. As discussed in section 7, our Enzian platform instance does not allow the FPGA to be booted independently, which is why the option CPU: off, FPGA: on is missing.

The last question we need to answer is how we need to apply our own State Generation procedure to these problem instances such that we do not actively prevent the worst-case performance from manifesting itself. Whether or not we see worst-case behaviour depends on the choices our procedure makes. More concretely, it depends on the sequence in which State Requirements are enforced and the sequence in which the individual State Possibilities are tried. With the roughly sixty conductors our Enzian platform features, exhaustively measuring performance on all such sequences is not feasible. We instead need to resort to randomisation.

Recalling our reasoning about why more advanced backtracking was needed in section 8.2.1, it is fairly clear that the performance of our naive backtracking mechanism will be poor for some such randomly chosen sequences. Thus, the question we wish to answer with our measurements boils down to the following:

Question 59. Are the problem instances P1 to P3 sufficiently well-conditioned that our State Generation procedure with advanced backtracking performs *reliably* enough that we can *empirically* bound its runtime on said instances?

The next paragraph discusses the measurement set-up in more detail.

Set-up Firstly, we want to discuss how the aforementioned randomness can be introduced into our implementation. Since State Possibilities are tried in-order in accordance with the likelihood heuristic discussion in section 8.2.1, we can randomize this order by simply shuffling the sequence in which they are stored in our platform instance. The order in which our implementation enforces requirements in Desired_States is governed by the platform attribute *sorted_wire_list*. Said contains an ordered list of all conductors, and in every iteration our procedure performs, the requirement associated with the conductor featuring the smallest index in said list is enforced. Thus, we can achieve randomisation by shuffling said attribute.

We perform three sets of runtime-measurements M1, M2 and M3 for every problem instance Pi $\in$ {P1, P2, P3}. We will describe the purpose of each measurement set below the detailed set-up given by the following list:

(M1) Repeat X times:

- Randomly shuffle State Possibilities and the *sorted_wire_list* as detailed above
- Configure the power states of CPU and FPGA according to Pi
- Measure the runtime of parametrized_state_search({}, flags) for each of the following flag combinations:

(a)

extend	True
all_solutions	False
use_z3	False
advanced_backtracking	True

(b)

extend	True
all_solutions	False
use_z3	False
advanced_backtracking	False

(c)

extend	True
all_solutions	-
use_z3	True
advanced_backtracking	-

(M2) Repeat Y times:

- Randomly shuffle State Possibilities and the *sorted_wire_list* as detailed above
- Configure the power states of CPU and FPGA according to Pi
- Measure the runtime of parametrized_state_search({}, flags) with flags defined as:

extend	True
all_solutions	False
use_z3	False
advanced_backtracking	True

(M3) Repeat Z times:

- Randomly shuffle State Possibilities and the *sorted_wire_list* as detailed above
- Configure the power states of CPU and FPGA according to Pi
- Measure the runtime of parametrized_state_search(**narrowed_isl_poss**, flags) with flags defined as:

extend	True
all_solutions	True
use_z3	False
advanced_backtracking	True

With M1, we wish gain a general overview of the performance of the different versions of our State Generation mechanisms. We especially wish to contrast the performance of our State Generation procedure with and without advanced backtracking, and whether there is any correlation between advanced and naive backtracking performing *poorly* on any

given sequence. We include a measurement of the Z3-based mechanism, which is independent of the sequence randomisation, to provide some baseline performance. Recalling the meaning of the flags as discussed in section 8.2.4, it follows that the measurements taken in M1 correspond to (a) the performance of our State Generation procedure with advanced backtracking, (b) the performance of our procedure with naive stack-based backtracking and (c) the performance of the Z3-based mechanism. In anticipation of the naive backtracking exhibiting bad performance, we choose a small X = 100.

As we have detailed in the last section, the worst case time required to find only a *single* solution on problems P1 to P3 is not sufficient to provide a worst-case runtime bound for all online static management problems. However, we are fairly convinced that the measurement set M2 provides some indication of the performance we can *expect* in general. We choose Y = 500.

Finally, M3 is supposed to to answer the question we have established in the last section. To provide more significant measurements, we are narrowing down what we refer to as *ISL possibilities*. On the Enzian, a total of four ISL6334d [20] producers is tasked with providing the supply voltage for both the ThunderX's as well as the FPGA's DRAM banks. The exact voltage the ISLs are supposed to supply is conveyed to them via their so-called *VID* inputs, 8 different logical inputs that select among 178 different available voltages. Naturally, each of these must be described by a dedicated State Possibility. If not narrowed down, these different possibilities cause our solution space to explode, thereby bloating the runtime of our State Generation procedure unnecessarily: Since the conductors providing these VID inputs are **not** shared in complex manners with logical inputs of other producers, any desired VID combination can be realised.

Therefore, we can safely narrow down these *ISL possibilities* in M3 without jeopardising our upper runtime-bound of online static management, provided that we also narrow down these possibilities for every such management request **prior** to handing it to our procedure.

To perform our measurements, we will make use of the *timeit* python library. Said provides a function *timeit*, which runs a function passed to it as many times as specified by the argument *number* and returns the total time measured. To provide more stable measurements, we set number = 3 for all method calls with the exception of the method calls in M1 that make use of naive backtracking.

Ideally, this runtime measurement would be conducted on the Enzian BMC to account for its limited computational power and memory. However, our implementation features non-standard dependencies such as the Z3 solver and its python wrapper and thus porting it to the BMC proper is a non-trivial task. Instead, we conduct our measurements on an Intel Core i7-6700k CPU @ 4.000GHz with 32GB DDR4 RAM.

Results In this paragraph, we examine the measurement data obtained from performing measurements M1 to M3. We collect a set of observations concerning the general nature of this data, which we will use in the next paragraphs to interpret our findings

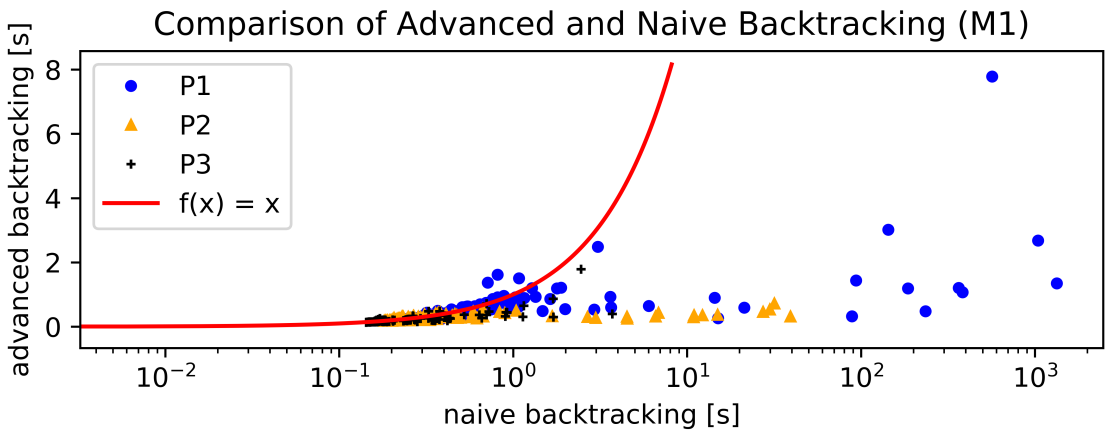


Figure 9.2: A performance comparison of the backtracking mechanisms. Every symbol in the plot contrasts the measured runtimes of advanced and naive backtracking on a particular randomised sequence. For reference, we have additionally plotted the function f(x) = x.

and draw appropriate conclusions.

As already mentioned in the last paragraph, our main motivation for measurement set M1 was contrasting the performance of advanced and naive backtracking. Since both mechanism versions are dependent on the randomisation of the Possibility and Conductor sequences, we have visualised the collected data in a scatter plot in figure 9.2. Every symbol in that figure relates a runtime measurement for advanced and naive backtracking on a particular such randomisation. Noting the log-scale of the x-axis, we make the following observations:

Observation 60. Based on figure 9.2, the runtime measurements of naive backtracking exhibit more variance than said of advanced backtracking.

Observation 61. Based on figure 9.2 and the plot of the function $f(x) = x$, advanced backtracking tends to perform better than naive backtracking, but is not always strictly better.

Additionally, M1 also included measurements of the State Generation mechanism based on the Z3-solver. Unlike our own State Generation procedure, the Z3 solver does not depend on the randomisation of Possibility and conductor sequences. Therefore, we have collected those measurements in a histogram in figure 9.3, rather than adding a third dimension to the scatter plot discussed above. Studying this histogram, we observe that:

Observation 62. According to figure 9.3, the performance of the Z3-based mechanism is very stable, also across problem instances.

The remaining measurement sets M2 and M3 focus on the performance of the advanced backtracking mechanism only. Since we aim to reason about worst-case performances,

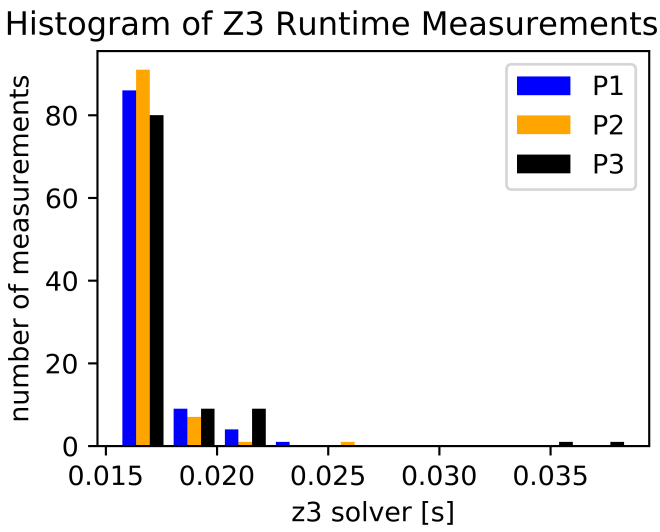


Figure 9.3: A histogram of runtime measurements of the Z3 solver on the different problem instances.

examining the general distribution of the collected measurement values seems most informative. We therefore visualise the remaining data as histograms.

For measurement set M2, we define an *outlier* to be a runtime measurement value of more than 2 seconds. Lacking more information about the true distribution of our runtime, this distinction by itself is pretty arbitrary. However, representing such *outliers* and more average values in separate histograms improves the clarity and readability of our diagrams, which can be found in table 9.2. We observe the following:

Observation 63. According to table 9.2, advanced backtracking tasked with finding only one solution takes less than 2 seconds in the majority of measurements.

Observation 64. Based on table 9.2, advanced backtracking tasked with finding only one solution for problem instances P1 to P3 exhibits substantial outliers, most notably one of 837.77 seconds.

Lastly, table 9.3 summarises the data we collected for measurement set M3. Unlike M2, splitting our measurements into outliers and normal values was not necessary to adequately represent the collected measurements. We observe the following:

Observation 65. According to table 9.3, advanced backtracking tasked with finding all solutions for the *reduced* problem instances P1 to P3 performs exhibits a tolerable runtime variance.

We conclude this paragraph with an observation that manifests itself across all measurement sets:

Histograms created from measurement set M2

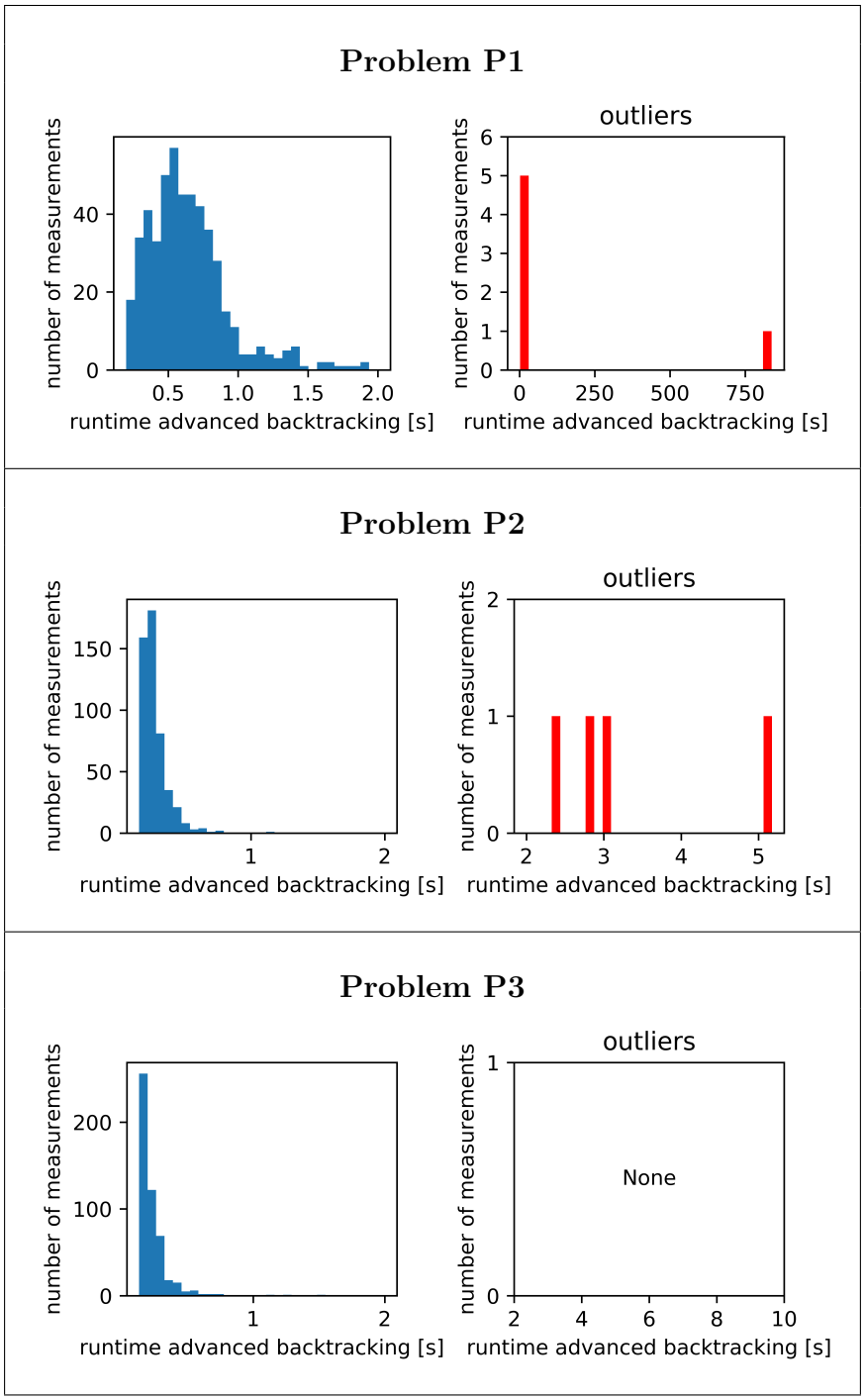


Table 9.2: Histograms of the data collected in M2: the performance of advanced backtracking when tasked with finding only one solution for problem instances P1 to P3.

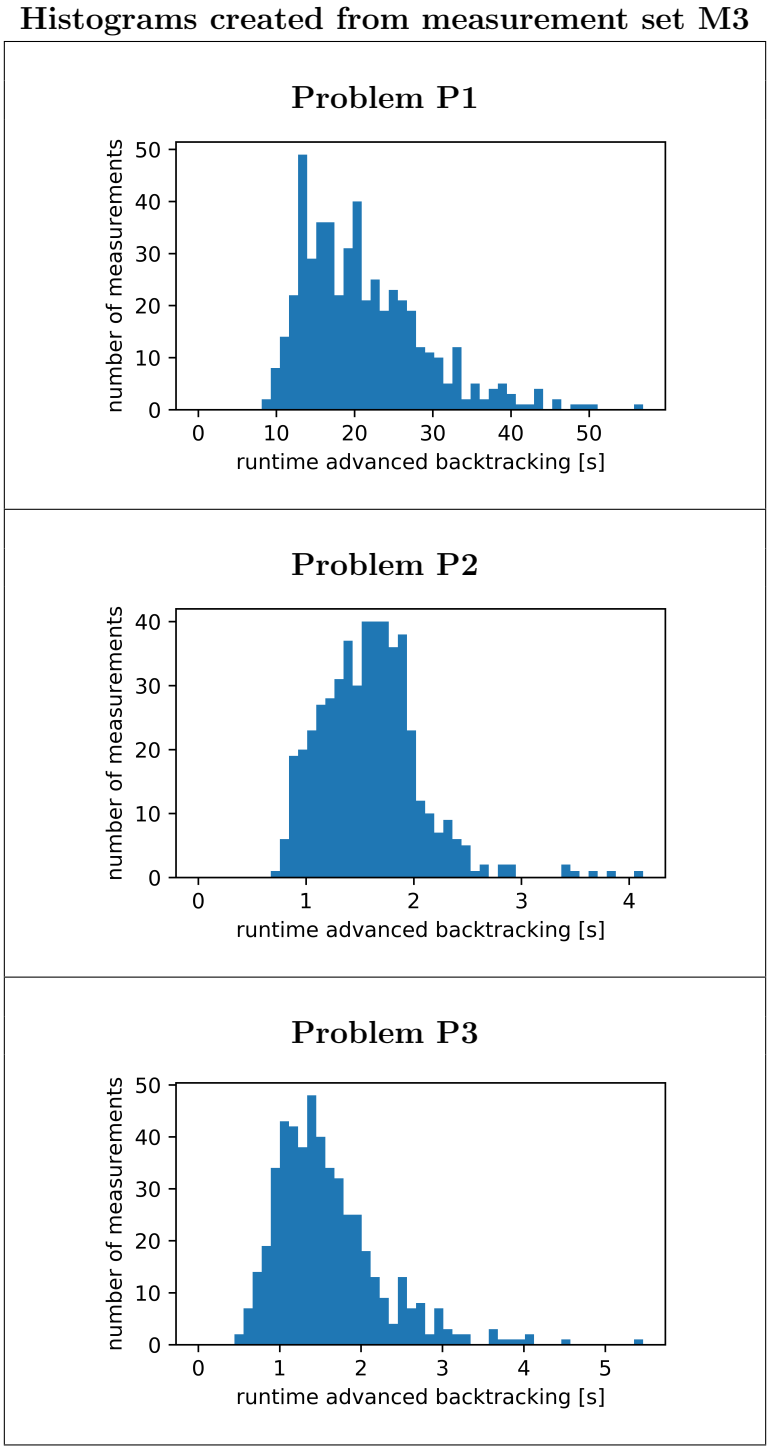


Table 9.3: Histograms of the data collected in M3: the performance of advanced backtracking when tasked with finding all solutions for restricted problem instances P1 to P3

Observation 66. Calls to our state generation procedure tasked with solving instance P1 exhibit the worst and most variable performance compared to the other problem instances.

Discussion In this paragraph, we try to interpret the observations we have made in the last paragraph.

The observations we have made in context with M1 (60 and 61) align with our expectations: Naive backtracking indeed exhibits bad performance on some random sequences. Furthermore, advanced backtracking features considerably better performance in most cases, but not always: This is not very surprising either, since the more advanced backtracking technique requires additional computations.

An interesting result is the stable performance of the Z3-based mechanism *across* problem instances, as noted in observation 62. Although we still cannot estimate the effect additional constraints will have on the Z3 solver's runtime, its stability across P1 to P3 implies that it most likely would not be too bad. Hence, the Z3-based mechanism might also be a viable candidate for an online static management solution for the Enzian platform.

The first observation we have made in context of M2 seems pretty promising: Most calls to our advanced backtracking performed by M2 took no longer than 2 seconds. However, the outlier of 837.77 seconds we have encountered for P1 is as unexpected as it is substantial. This immediately implies that we cannot in good conscience provide a runtime bound for calls to our advanced backtracking procedure to provide a *single* solution for problems P1 to P3. We recall observation 57, where we had established that the worst-case runtime of finding a *single* solution is a strict lower bound of the worst-case runtime of finding *all* solutions for problem instances P1 to P3. From this immediately follows that we also cannot provide an empirical bound of the worst-case runtime of **general** online static management problems, as was the original goal of this evaluation.

In light of this, the tolerable variance in runtime (see observation 65) for the data collected in M3 seems purely coincidental. This is not necessarily the case: Measurement set M3 is not a direct extension of M2 that is tasked with finding *all* solutions rather than only one. We have additionally also narrowed down the *ISL possibilities*, as defined in the beginning of this section.

We claim that the runtime of the outlier encountered in M2 can be largely attributed to not restricting said ISL possibilities. Intuitively, this seems quite likely: our advanced backtracking mechanism is still fairly simplistic and only *conservatively* approximates the State Possibility choice responsible for the incompatible conductor states. It may thus very well be the case that it ends up naively trying out all of the 178 different ISL possibilities because advanced backtracking fails to recognise that the choice that prevents our procedure from finding a solution was already made beforehand.

To evaluate the likelihood of our claim, we conduct an additional set of measurements

M4:

(M4) Repeat 500 times:

- Randomly shuffle State Possibilities and the *sorted_wire_list* as detailed above
- Configure the power states of CPU and FPGA according to Pi
- Measure the runtime of the following method calls:
- parametrized_state_search({}, flags)
- parametrized_state_search(narrowed_isl_poss, flags)

  With flags defined as follows:

extend	True
all_solutions	False
use_z3	False
advanced_backtracking	True

In essence, M4 repeats measurement set M2 but additionally contrasts the obtained runtime with said of a call with narrowed ISL Possibilities.

Similar to M1, we visualise our findings using a scatter plot (see figure 9.4). Studying this plot, we observe that the restriction of the problem sets eliminates all *outlier* measurements (> 2s) that would appear when solving the original problems P1 to P3. Furthermore, but unsurprisingly, we can see that the performance on the restricted problem set is strictly better. This is the case because as detailed in the beginning of this section, the restriction is *benign* since all ISL possibilities are achievable.

Hence, based on the lack of substantial outliers observed for M3, we provide an empirical worst-case runtime bound for online static management problems with **restricted ISL states** of 56.90 seconds, which corresponds to the slowest runtime observed in M3.

To conclude our evaluation, we provide a possible explanation for observation 66, which stated that of all problem instances, P1 always featured the worst and most variable performance. P1 corresponds to both CPU and FPGA being powered on and therefore all ISL producers of the platform being active. Consequently, if not restricted (as was the case for measurements M1 and M2), the negative effect these possibilities might have in context of backtracking are strongest in P1. In case of M3, where we dealt with restricted problem instances, the fact that P1 was the most difficult to solve can be attributed to the number of feasible solutions:

Problem instance	**Number of solutions**
Restricted P1	256
Restricted P2	6
Restricted P3	8

Table 9.4: The number of solutions for restricted P1 to P3

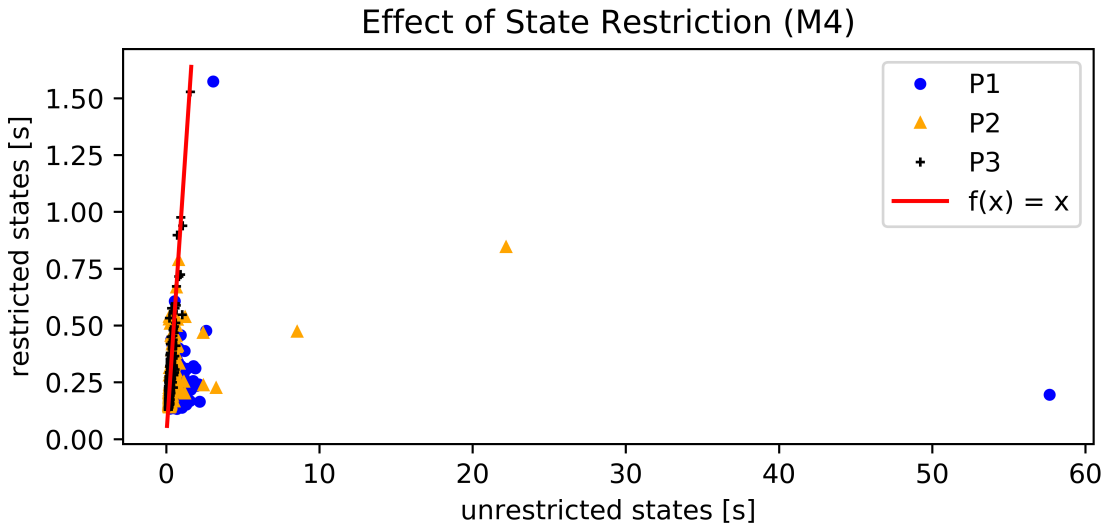


Figure 9.4: A performance comparison of advanced backtracking with and without restricted ISL states (according to M4).

As can be seen in the above table, reduced P1 features the largest number of feasible solutions. Since M3 is tasked with retrieving all possible solutions, this does manifest itself in the measurements for P1.

9.1.2 Offline Static Management

As mentioned in the beginning of the section, for our mechanisms to be sufficient for *offline* static management it suffices to show that their runtime is *manageable* for the Enzian platform instance. We have already received some indication that this is the case for the tied-together State and Sequence Generation mechanisms in the last evaluation. Therefore, this evaluation focuses on the performance of the Consumer Demand generation.

Quite generally, 6 different combinations of consumer transitions exist on the Enzian platform, namely:

Problem designation	Consumer	Initial consumer state	Final consumer state
P1	CPU	Powered off	Powered on
	FPGA	Powered off	Powered on
P2	CPU	Powered off	Powered on
	FPGA	Powered off	Powered off
P3	CPU	Powered on	Powered on
	FPGA	Powered off	Powered on
P4	CPU	Powered on	Powered off
	FPGA	Powered off	Powered off
P5	CPU	Powered on	Powered on
	FPGA	Powered on	Powered off
P6	CPU	Powered on	Powered off
	FPGA	Powered on	Powered off

Table 9.5: Overview of problem instances P1 to P6

As discussed in more detail in the next paragraph, we will perform runtime measurements for each of the above transitions. In contrast to the requirements of *online* static management, it suffices if we detect manageable runtime for at least one, fixed sequence of State Possibilities and Desired_States-enforcement. For our evaluation, we will therefore stick to the deterministic initialisation values given to all such sequences.

Set-up The general environment and test methodology is identical to the last evaluation. For every problem Pi ∈ {P1, ..., P6} we perform measurements as follows:

- Configure the power states of CPU and FPGA according to Initial(Pi)
- Measure the runtime of the method call stateful_node_update(Final(Pi), flags) with flags defined as follows:

extend	True
all_solutions	False
use_z3	False
advanced_backtracking	True

Results The obtained measurements can be found in table 9.6. None of the problem instances took longer than 3 seconds to solve. All of the measured runtimes, which can be found in table 9.6, are within at most 3 seconds.

Problem instance	Measured runtime [s]
P1	2.7
P2	1.3
P3	1.7
P4	0.4
P5	1.3
P6	1.5

Table 9.6: Measurements obtained for the six different combinations of consumer transitions possible on the Enzian platform

Discussion The required 3 seconds are surely tolerable for the generation of offline static management solutions. What is interesting to note (in comparison with the previous evaluation) is that the default Possibility and Conductor sequences seem to result in very good performance. This is not entirely by design: While the Possibility sequences do follow the *likelihood* heuristic as detailed in section 8.2.1, the Conductor sequence still features the order in which the conductor connections were specified when constructing the Enzian platform description.

Furthermore, the measured times quite nicely reflect the complexity of the different transitions: For instance the shut-down sequence for the FPGA is a lot more complicated than said of the CPU: consequently, P4 can be solved more quickly than P3.

9.2 Comparison to existing management solution

As already mentioned in section 8.2.4, the existing management solution has been developed further during the writing process of this thesis. For the purposes of this evaluation, we therefore refer to a meanwhile slightly outdated version. At the time, said management solution was focused on the correct bring-up and shut-down of both CPU and FPGA required for testing the correctness of the new Enzian board passes.

In this experiment, we want to see how closely our mechanisms can reproduce the bring-up sequence implemented in this existing management solution [21]. To achieve a meaningful comparison, we need to make some slight changes to our implementation. The reason for this is the following: The mentioned bring-up sequence is *mostly* booting CPU and FPGA as sequentially as possible. Our implementation of Consumer Demand generation is returning only one possible interleaving (which is sufficient under the assumption that Sequence Generation does not fail) and is currently tweaked to prefer *concurrent* interleavings (i.e. speaking in terms of the generated DP-table, as *diagonal* solutions as possible), since said require fewer static management actions. We will thus, for the purposes of this evaluation, change this tweaking to prefer CPU-before-FPGA instead. Visualised with the *Feasible DP table* our mechanism is constructing for the bring-up of both CPU and FPGA, this looks as follows:

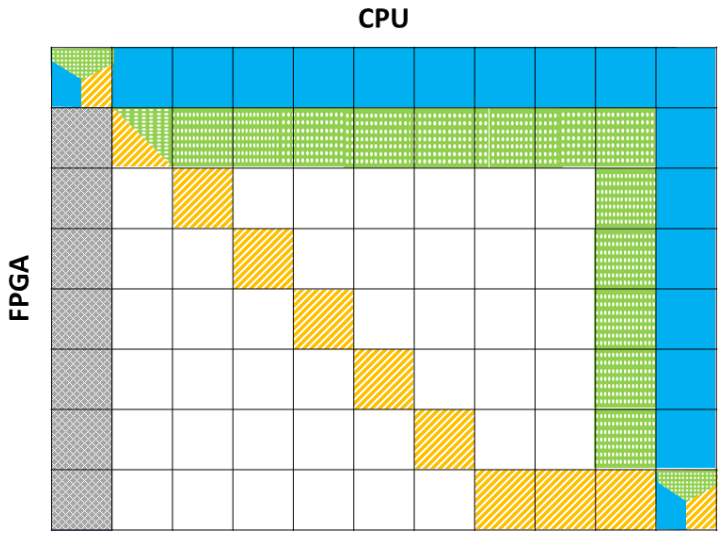


Figure 9.5: A visualisation of the different interleavings: With the interleaving our implementation normally prefers in orange (striped), said we encourage with our changes in blue and the one implemented in the existing management solution in green (dotted). Grey field with grid pattern mark impossible states.

Furthermore, we will task our **stateful_node_transition** method to compute all solutions. Recalling the meaning of the different flags presented in 8.2.4, this will have the following effect: The Stable Platform state of every transition step corresponds to will be chosen in a manner that requires the fewest changes from the previous transition step. This more closely mirrors how a human would approach the generation of such boot sequences: In every step, perform exactly the changes necessary. By asking for *all* solutions, we encounter the same issue as in the first evaluation: The wealth of ISL Possibilities prevent us from finding *all* solutions in reasonable time. We will therefore once more restrict these possibilities, which in this case requires a modification our consumer descriptions.

With that said, what remains to discuss are the means with which we can evaluate the two solutions. We are going to do so in two steps:

(S1) Determining the similarity of the set of commands present in each sequence.

(S2) Determining the similarity of the ordering of the commands.

If we wish to obtain a meaningful result, the comparison of the ordering is not as trivial as one might think at first glance. Of course, we could simply evaluate the number of transpositions necessary to transform one command sequence into the other. However, as discussed in section 6.3.4, our mechanisms generate command sequences by topologically sorting a global Event Graph. In general, a topological order is not unique; this is especially true for Event Graphs of the Enzian platform: The events of producers on

the same hierarchy level (which are depicted in figure 7.1) can generally be ordered arbitrarily. Consequently, a lot of information about the possible sequences is lost when our mechanisms sort the Event Graph.

Since we wish to determine how closely our mechanisms can *reproduce* the manually constructed boot sequence, we will therefore base the comparison of the command orderings on the internal Event Graph representations rather than the final sequence.

Set-up After applying the aforementioned changes to our implementation, we gather our evaluation data as follows:

- Set the power states of both FPGA and CPU to Powered-down.
- Run apply_changes({}, default_flags)
- Extract the command sequence C_{gen} and event graphs generated by the command: stateful_node_transition({CPU: "Powered_on", FPGA: "Powered_on")

Note that the call to apply_changes is necessary to ensure that our platform features the correct initial virtual state. Furthermore, according to the operation of our Consumer Demand Generation mechanism, a separate global event graph is generated for every step in the generated interleaving, and we thus obtain a set of Event Graphs {G1, ..., Gn}.

For the evaluation of the similarity of the two command sets (S1) we then proceed as follows: Let S_{gen} and S_{man} denote the generated and manual command sets respectively. We manually construct the following relations on $S_{gen} \times S_{man}$:

Syn: {(a, b) | a $\in S_{gen}$, b $\in S_{man}$ and a, b are syntactically equal}
Sem: {(a, b) | a $\in S_{gen}$, b $\in S_{man}$ and a, b are semantically equivalent}

Whereby two commands are syntactically equal if their corresponding strings are equal. For the purposes of this evaluation, we informally define *semantic equivalence* to only consider the effects on the platform. As an example, consider the helper function

check_voltage(identifier, f, min, max)

that operates as follows: The monitoring function f is called repeatedly until the value it returns is between (min, max). If this does not happen within a certain time period, check_voltage returns an exception. $Identifier$ is a human-readable string used for logging and to generated meaningful error messages.

For our purposes, we identify the following two calls to check_voltage to be semantically equivalent:

```
1 check_voltage("foo", f, 5, 7)
2 check_voltage("bar", f, 2, 4)
```

That is because the monitoring function f is the same in both calls, which renders the two commands indistinguishable from the perspective of the platform: The state of the same conductor is being queried in the exact same fashion and only the subsequent processing of the return value, which is transparent to the platform, differs. In contrast to this, the following two commands that set the state of a gpio pin "pin" would not be considered the same, since the platform is supposed to react differently to the two commands and must therefore be able to distinguish them:

```
1 gpio.set_value("pin", False)
2 gpio.set_value("pin", True)
```

What we want from this relaxed definition of semantic equivalence is that monitoring commands which only differ slightly in their definition of error margins, for instance (4.51, 5.49) and (4.5, 5.5), are still considered equivalent. Unfortunately, we also classify monitoring commands with different target values as semantically equivalent; this is, however, not such a huge problem as it might seem at first glance: For a check of a specific conductor state to make sense, said state must have been caused intentionally beforehand. Hence, differing monitoring target values must also be reflected elsewhere in the command sequence.

With our definition of semantic equivalence in place, we need to make one final remark about the nature of our defined relations **Sem** and **Syn**: For our command sequences, it holds that Syn $\subseteq$ Sem. This is not true for arbitrary Python programs: two command strings appearing in different programs might be equal, yet if their execution contexts differ and their evaluation is context sensitive, they might not be semantically equivalent. That Syn $\subseteq$ Sem is the case of our command sequences is quite important, since otherwise a comparison of the command sets would be nonsensical in the first place.

With all that in mind, we will compare the similarity of command sets by discussing each of the following sets:

(A) the command pairs appearing in **Syn**: commands that appear in exactly the same form in both sequences.

(B) the command pairs appearing in **Sem** but not in **Syn**: command pairs that are syntactically different yet have the same semantic meaning.

(C) the commands in S_{man} that do not appear in **Sem**: commands of S_{man} that do not semantically correspond to any command in S_{gen}

(D) the commands in S_{gen} that do not appear in **Sem**: commands of S_{gen} that do not semantically correspond to any other command in S_{man}

Figure 9.6: Definitions of sets of interest in the comparison of S_{gen} and S_{man}

Comparing the ordering of the two sequences is a bit more involved. In preparation

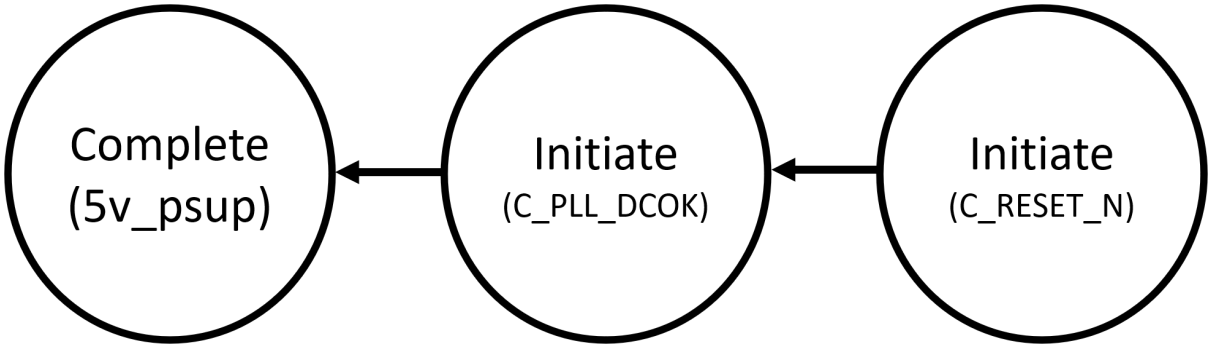
for this step, we identify the Complete and Initiate events appearing in the manually generated sequence, as well as the event relations given by their fixed order. We thus obtain an event graph G defined by the manual sequence. As an example, consider the following snippet of manually generated code:

Listing 9.1: a code snipped of the manually generated management solution [21], whereby function f was abstracted for better readability

```
1 check_voltage('FPGA:5V_PSUP', f, 4.75, 5.25)
2 gpio.set_value('C_PLL_DCOK', False)
3 gpio.set_value('C_RESET_N', False)
```

Whereby check_voltage and gpio.set_value have the effects as discussed above. In line 1, the identifier "FPGA:5V_PSUP" passed to check voltage provides the name of the conductor whose state f is checking. Therefore, we associate the event **Complete(5v_psup)** with line 1. On the Enzian, the conductors connected to gpio pins "C_PLL_DCOK" and "C_RESET_N" are named equivalently, hence we associate events **Initiate(C_PLL_DCOK)** and **Initiate(C_RESET_N)** with lines 2 and 3 respectively.

In summary, the Event Graph G we construct for the above snipped would be of the following form: (whereby the meaning of the directed edges is defined as in section 5.5.4):



The question that remains is how we are going to compare the event graphs $\{G1, ..., Gn\}$ extracted from our procedure to graph G of the manual sequence. Intuitively, we would like to combine $\{G1, ..., Gn\}$ to a global event graph G_{Gen} by adding the necessary edges that enforce the ordering between these graphs given by the interleaving steps. Then, we could extend G_{Gen} with the edges defined by G, and check if the resulting graph is still acyclic. If this were the case, this would imply that the unique topological order dictated by G corresponds to a specific topological order of G_{Gen}. Unfortunately, recalling figure 9.5 from the previous paragraph, we can already tell that this will not be the case, since the interleaving the manually generated solution is based on differs from the interleaving generated by our procedure.

We can work around this by instead extending every event graph in {G1, ..., Gn} with G and checking for cycles. It is important to note, however, that G is not necessarily acyclic itself: A consumer might for instance ask for a logical signal s to be asserted during some

step x in the middle of its transition sequence, and deasserted during the remaining steps. This would of course cause the event Initiate(s) to appear several times in G, which would, due to the strict order of G, inherently result in a cycle. In the case of the Enzian, this is not an issue. However, we must nevertheless carefully exclude any commands from the manually generated sequence, that (similar to our call to apply_changes) ensure that the platform is in a known and correct initial state. For instance, the two gpio sets in the code snippet above (listing 9.1) must not be added to G: they ensure that these two gpio pins are deasserted during the boot sequence, as required by the ThunderX.

To determine if the command ordering given by the manual sequence could have been found by our implementation, we therefore perform the following checks:

(C) For i from 1 to n:
Gi $\leftarrow Gi \cup G$
print(is_acyclic(Gi))

Results We have listed both command sequences C_{gen} and C_{man} in listings A.2 and A.1 in the appendix, along with table A.1, which specifies relations Syn and Sem by referencing the corresponding line numbers in the command sequences, as well as the corresponding event if applicable.

To summarise the result of the comparison of the command sets, we provide the cardinality sets of interest (A) to (D) which we have identified in the previous paragraph:

Set designation	Cardinality
(A)	40
(B)	8
(C)	5
(D)	12

Table 9.7: Summary of command set comparison in appendix table A.1

Hence, there is an exact correspondence (A) between most of the commands in the two sequences. We will discuss the deviations given by sets (B) to (D) in more detail in the next paragraph.

As mentioned above, comparing the similarity of the ordering of the two sequences (S2) required us to construct the Event Graph G of the manual sequence. The table A.1 also includes the event associated with the commands in the manual sequence, but we refer to the annotated_manual_sequence.py file in the code repository of this thesis for a more convenient representation.

We have found that all checks (C) as defined in the last paragraph evaluated to **True**, meaning that combining each Event Graph in {G1, ..., Gn} with G resulted in an acyclic graph.

Discussion As mentioned above, we first wish to reason about the deviations of our two command sets as indicated by sets (B) to (D):

(B) Recalling the definitions given by list 9.6, (B) specifies pairs of commands that are semantically equivalent yet differ syntactically. In table A.1 of the appendix, we have provided the reason for lacking syntactic equality for each pair in (B): In every case, said reason is either "helper function" or "margins". The former means that the manual solution is calling a helper function at this point to make the sequence more readable, whereas the automatically generated solution directly replaces such functions with their definition. The latter refers to differing error margins passed to the function *check_voltage*. As discussed in the context of our definition of semantic equivalence, these error margins could differ arbitrarily, which they however do not in our case; the median value, which generally denotes the target voltage if we allow for equal deviations above and below is the same for every pair in (B) with differing margins. What must be noted at this point, however, is that the margins assigned by our mechanisms are not trustworthy: Rather than accurately calculating the expected error according to the various data sheets, we have simply used a fixed tolerance of 5%.

(C) Set (C) defines the commands appearing in the manually generated sequence that do not semantically correspond to a command in S_{Gen}. As discussed in more detail in the appendix, this has one of two possible reasons: Either the corresponding conductors are not modelled by our platform instance, or the commands serve to assert a precondition of the bootstrapping process. In the latter case, the generated sequence also features said commands, but they are produced by preceding the call to *apply_changes* and therefore not considered in this evaluation.

(D) Set (D) contains commands in S_{Gen} that lack a semantically equivalent command in S_{man}. We can once more identify two possible causes for this: Either the command in question is redundant or concerned with the supply of the DDR4 DRAM banks of CPU or FPGA, which had not yet been fully implemented in the manually generated solution. We again refer to the appendix for a more detailed discussion.

In conclusion, the two command sets are not exactly equal but we have also not found any major discrepancies. Thus, the comparison of their ordering, which we discuss next, is not rendered nonsensical.

As stated in the results-paragraph, the checks (C) we performed all returned True. This means that within each transition step, the ordering of the commands our Sequence Generation mechanisms is trying to enforce is not in conflict with the ordering provided by the manually generated sequence. This means that ultimately, our mechanism is sufficiently *complete* to reproduce an almost equivalent sequence (modulo differing commands) if we were to adjust the transition interleaving to match said of the manual sequence.

Does this imply anything about the correctness of our Sequence Generation mechanism? No, unfortunately not; in theory, our Sequence Generation mechanism could also have generated Event Graphs $\{G1, ..., Gn\}$ without any edges, i.e. Event Graphs that do not

specify any restrictions. The predicate Acyclic(Gi ∪ G) for our acyclic G would then be vacuously true. However, we strongly believe that our generated sequence should at least in theory and to the extent of our knowledge be able to correctly boot the CPU and FPGA on the Enzian platform. We say "to the extent of our knowledge" since we would not be surprised if there existed additional issues that need to be considered that we are not aware of.

For instance, the choice of interleaving the manual sequence is based on is clearly deliberate: The transition sequences of both CPU and FPGA require their respective clock signals to be configured in their first transition step and demand that these signals are stable (have locked onto the target frequencies) as part of the last transition step. The manual solution boots CPU and FPGA sequentially, with the exception that all the clock configurations are done as early (a) and all the checks of stability as late (b) as possible (see figure 9.7). It follows that the phase locking of clock generators probably takes a considerable amount of time compared to the rest of the operations. This is taken into account by the manual boot sequence, but our current platform model lacks the expressiveness to encode additional timing information into the consumer boot sequences.

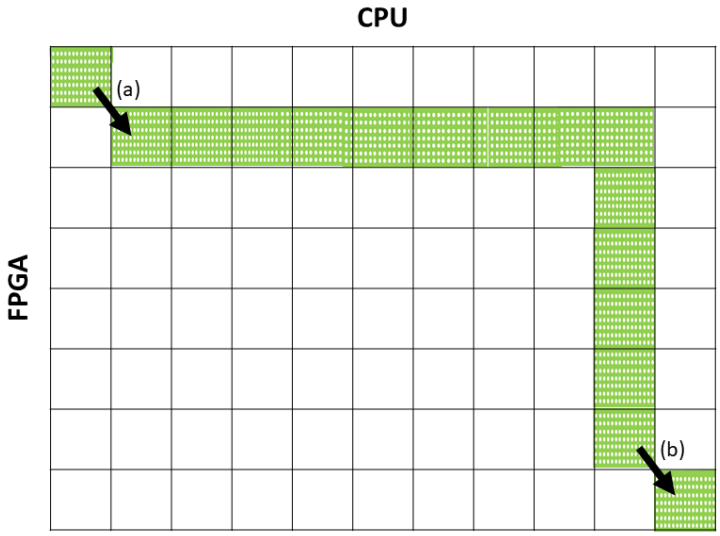


Figure 9.7: The manual boot sequence with the clock configuration step labelled with (a) and the phase-locking check step labelled with (b)

9.3 Conclusion

In this chapter, we have attempted to evaluate the feasibility of our management solution for the Enzian platform. We have been able to establish an empirical worst-case runtime bound for online static management for *restricted* problem instances. However,

due to said being set to 56.90 seconds, the feasibility of our approach for online static management remains questionable.

In spite of this, as has been demonstrated by our second evaluation, our approach is more than sufficiently efficient to find offline static management solutions that can then be hardcoded into a BMC implementation.

Apart from the efficiency of our mechanisms, we have also attempted to compare a generated sequence to the existing manually created bootstrap sequence. While these sequences do differ in some aspects, we have concluded that our mechanisms could be made to almost reproduce to given manual sequence. We have also observed that this by itself does not imply anything about the correctness of our mechanisms.

Chapter 10

Conclusion

This chapter concludes our thesis. In a first section, we summarise our results. Finally, we outline possible future work in section 10.2.

10.1 Summary

In this thesis, we have discussed a model-mechanism based approach to static power and clock management. We started by developing a general perspective on what power and clock management entails on modern platforms and as well as an idea of when it can be considered *correct*.

We have used this knowledge to devise a model that captures correct platform behaviour. This entails the stable states our platform can assume as well as correct transitions between such states. In connection with such transitions, we have introduced the concept of events and were able to further refine our ideal vision of correct static management. We have not been able to integrate bus operability to a satisfactory degree and have discussed the situations in which our model would fail to adequately capture a transition.

In chapter 6, we have discussed the mechanisms necessary to extract static management actions from our model. We have seen that in their general form, all problems we are required to solve are NP-hard. We have introduced appropriate algorithms and, in case of Sequence Generation, an approximation that made the problem more manageable.

As a proof of concept, we have modelled the Enzian platform and implemented our mechanisms. We have discussed why we consider the Enzian to be well-designed and have given some general platform modelling advice. We have made an effort to increase the efficiency of our mechanism implementations, occasionally by assuming that our modelling advice has been followed.

In our evaluation (chapter 9), we have shown that our model-mechanism based approach is feasible for the Enzian platform. Due to the lack of literature on platform-level power and clock management, we have based some of our perspective on static management on observations made in context of the Enzian. Consequently, we cannot be sure that our model is general enough to capture the behaviour of other platforms, but we think it is

likely. Therefore, we have found a more generally applicable solution to static power and clock management.

However, any gain in generality is automatically associated with a loss of specialisation: By expressing a platform's behaviour using our general model, we automatically give up on any more advanced features our platform components might have. For instance, the MAX15301 provides a non-volatile memory where certain configurations could be stored, which could be used to change the default voltage provided after a power-cycle on-the-fly.

But in some cases, a static management solution relying on the potentially low-quality flash memory installed in a MAX15301 may not be not desirable. Perhaps the time of the designer of such regulators would be better used to provide a more formal specification instead of additional fancy features.

10.2 Future Work

As already mentioned in the introduction, we have not been able to formally verify the correctness of our mechanisms. This is therefore the first item we wish to mention in this section about future work.

With the formal verification in place, we would be able to generate provably correct static management actions for any platform our model can express. It would thus be interesting to evaluate how well our model generalises to other platforms. We hypothesise that due to the need to reduce production and material costs, commercially available platforms might not exhibit such a clean and optimal design as the Enzian platform does. We thus think it likely that such platforms would push our model to its limits.

Furthermore, recalling our definition of scope in chapter 4, we have only addressed a part of power and clock management. To truly provide an alternative to today's point solutions, we would need to extend our approach to include dynamic management. As previously discussed, dynamic management is vital to ensure that no exceptional situation endangers the health and integrity of our platform's components.

Currently, our implementation does not exhaust the full potential of being able to generate *all* stable platform states that could be adopted next. We could introduce additional criteria to encourage the selection of a good solution, for instance by increasing the efficiency of linear regulators as given by ([10], p. 120).

Last but not least, our current implementation lacks integration with any of the interfaces used for communication between BMCs and consumers. Potentially, if our approach were used as an online management solution, the dynamic generation of solutions might enable us to allow consumers more fine-grained control over the state of their input conductors than provided by the pre-defined power states of the ACPI interface.

Bibliography

[1] Z3prover. https://github.com/Z3Prover/z3. Online. Accessed 2020-08-23.

[2] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. Formalizing memory accesses and interrupts. *Electronic Proceedings in Theoretical Computer Science*, 244:66–116, Mar 2017. ISSN 2075-2180. doi: 10.4204/eptcs.244.4. URL http://dx.doi.org/10.4204/EPTCS.244.4.

[3] L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli. Policy optimization for dynamic power management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):813–833, 1999.

[4] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000.

[5] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, Christoph M. Wintersteiger, Zhiming Liu, and Zili Zhang. Programming z3. In *Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures*, pages 148–201. Springer International Publishing, Cham, 2019. ISBN 978-3-030-17601-3. doi: 10.1007/978-3-030-17601-3_4.

[6] Anthony J. Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the security issues surrounding lights-out server management. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, page 10, USA, 2013. USENIX Association.

[7] cavium. *Cavium ThunderX CN88XX, Pass 2 Hardware Reference Manual*, 2017.

[8] UEFI forum. *Advanced Configuration and Power Interface (ACPI) Specification, Version 6.3*, 2019. URL https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf. Online. Accessed 2020-07-31.

[9] Jessie Frazelle. Opening up the baseboard management controller. *Queue*, 17:5–12, 10 2019. doi: 10.1145/3371595.3378404.

[10] Corey Gough, Ian Steiner, and Winston Saunders. Platform power management. In *Energy Efficient Servers: Blueprints for Data Center Optimization*, pages 93–151. Apress, Berkeley, CA, 2015. ISBN 978-1-4302-6638-9. doi: 10.1007/978-1-4302-6638-9_4.

[11] Texas Instruments. *Understanding the I²C Bus, Application Report SLVA704*, 2015. URL https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1593018199437&ref_url=https%253A%252F%252Fwww.google.com%252F. Online. Accessed 2020-06-24.

[12] Maxim Integrated. *DDR Memory-Termination Supply, Application Note 1857*, 2003. URL https://pdfserv.maximintegrated.com/en/an/AN1857.pdf. Online. Accessed 2020-04-08.

[13] Maxim Integrated. *MAX15301. InTune Automatically Compensated Digital PoL Controller with Driver and PMBus Telemetry, Document Revision 4*, 2013. URL https://datasheets.maximintegrated.com/en/ds/MAX15301.pdf. Online. Accessed 2020-07-28.

[14] Intel. *Intelligent Platform Interface Specification Second Generation, Document Revision 1.1*, 2013. URL https://www.intel.com/content/www/us/en/products/docs/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html. Online. Accessed 2020-07-31.

[15] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1), February 2014. ISSN 0734-2071. doi: 10.1145/2560537. URL https://doi.org/10.1145/2560537.

[16] Linda Lua. *Clock Tree 101*. SILICON LABS. URL https://www.mouser.com/pdfdocs/clock-tree-101-timing-basics.pdf. Online. Accessed 2020-06-25.

[17] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. Redleaf: Towards an operating system for safe and verified firmware. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 37–44, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367271. doi: 10.1145/3317550.3321449. URL https://doi.org/10.1145/3317550.3321449.

[18] International Rectifier. *IR5336, 50AIntegratedPowIRstage, V3.1*, July 2014.

[19] Andrew Regenscheid. *Platform Firmware Resiliency Guidelines, Nist Special Publication 800-193*, 2018. URL https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf. Online. Accessed 2020-08-02.

[20] Renesas. *ISL6334D VR11.1, 4-Phase PWM Controller with Phase Dropping, Droop Disabled and Load Current Monitoring Features, Document Revision 4.00*, APR 2016. URL https://www.renesas.com/us/en/www/doc/datasheet/isl6334d.pdf. Online. Accessed 2020-08-24.

[21] Enzian BMC Power Management Tools Repository. https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bmc-powermgmt/-/tree/master. Online. Accessed 2020-06-03. Commit SHA 7a2e2c30.

[22] Daniel Schwyn. Hardware configuration with dynamically-queried formal models. Master's thesis, ETH Zurich, October 2017. URL http://www.barrelfish.org/publications/ma-schwyda-hwconf.pdf.

[23] Lattice Semiconductor. *ispPAC-POWR1220AT8, Data Sheet DS1015*, 2012. URL https://www.latticesemi.com/-/media/LatticeSemi/Documents/Solutions/Packaging-Solutions/ispPAC_POWR1220AT8-Data-Sheet1015.ashx?la=en. Online. Accessed 2020-07-20.

[24] ON Semiconductor. *NCP51400, NCV51400, Document Revision 6*, June 2020. URL https://www.onsemi.com/pub/Collateral/NCP51400-D.PDF. Online. Accessed 2020-08-23.

[25] The Enzian Team. Enzian: a research computer built by the systems group at eth zurich. URL http://enzian.systems. Online. Accessed 2020-07-28.

[26] Pai-Chuan Yang, Hong-Tzer Yang, and Ching-Lien Huang. Solving the unit commitment problem with a genetic algorithm through a constraint satisfaction technique. *Electric Power Systems Research*, 37(1):55 – 65, 1996. ISSN 0378-7796. doi: https://doi.org/10.1016/0378-7796(96)01036-X. URL http://www.sciencedirect.com/science/article/pii/037877969601036X.

Appendix A

Evaluation 3

This section contains the command sequences along with additional evaluation data discussed in section 9.2.

A.1 Manually generated sequence

Listing A.1: The manually generated boot sequence, taken from [21]

```
psup_on()
wait_for('PSUP_PGOOD', lambda: "IN1" in power.device_read("pac_cpu", "input_status"), True, 1)
check_voltage('CPU:12V_CPU0_PSUP', lambda: read_pac_adc("pac_cpu", "VMON1"), 4.71, 5.19)
check_voltage('CPU:5V_PSUP', lambda: read_pac_adc("pac_cpu", "VMON2"), 4.75, 5.25)
check_voltage('CPU:3V3_PSUP', lambda: read_pac_adc("pac_cpu", "VMON3"), 3.135, 3.465)
check_voltage('FPGA:12V_CPU1_PSUP', lambda: read_pac_adc("pac_fpga", "VMON1"), 4.71, 5.19)
check_voltage('FPGA:5V_PSUP', lambda: read_pac_adc("pac_fpga", "VMON2"), 4.75, 5.25)
gpio.set_value('C_PLL_DCOK', False)
gpio.set_value('C_RESET_N', False)
gpio.set_value('B_OCI2_LNK1', False)
gpio.set_value('B_OCI3_LNK1', False)
program_clock_main()
program_clock_cpu()
program_clock_fpga()
program_ir3581()
wait_for('B_CLOCK_BLOL', lambda: gpio.get_value('B_CLOCK_BLOL'), True, 10)
enable_pac_out('pac_cpu', 'OUT6')
check_voltage('CPU:VDD_CORE', lambda: read_pac_adc("pac_cpu", "VMON4"), 0.94, 0.96)
enable_pac_out('pac_cpu', 'OUT7')
check_voltage('CPU:0V9_VDD_OCT', lambda: read_pac_adc("pac_cpu", "VMON5"), 0.87, 0.93)
enable_pac_out('pac_cpu', 'OUT8')
check_voltage('CPU:1V5_VDD_OCT', lambda: read_pac_adc("pac_cpu", "VMON6"), 1.425, 1.575)
enable_pac_out('pac_cpu', 'OUT9')
check_voltage('CPU:2V5_CPU13', lambda: read_pac_adc("pac_cpu", "VMON7"), 2.375, 2.625)
enable_pac_out('pac_cpu', 'OUT10')
check_voltage('CPU:2V5_CPU24', lambda: read_pac_adc("pac_cpu", "VMON8"), 2.375, 2.625)
enable_pac_out('pac_cpu', 'OUT11')
enable_pac_out('pac_cpu', 'OUT12')
enable_pac_out('pac_fpga', 'OUT6')
check_voltage('FPGA:UTIL_3V3', lambda: read_pac_adc("pac_fpga", "VMON3"), 3.135, 3.465)
enable_pac_out('pac_fpga', 'OUT9')
enable_pac_out('pac_fpga', 'OUT13')
check_voltage('FPGA:VCCINTIO_BRAM_FPGA', lambda: read_pac_adc("pac_fpga", "VMON10"), 0.873, 0.923, 1)
enable_pac_out('pac_fpga', 'OUT15')
check_voltage('FPGA:VCC1V8_FPGA', lambda: read_pac_adc("pac_fpga", "VMON11"), 1.71, 1.89)
enable_pac_out('pac_fpga', 'OUT16')
check_voltage('FPGA:SYS_1V8', lambda: read_pac_adc("pac_fpga", "VMON12"), 1.71, 1.89)
enable_pac_out('pac_fpga', 'OUT8')
check_voltage('FPGA:SYS_2V5_13', lambda: read_pac_adc("pac_fpga", "VMON5"), 2.375, 2.625)
enable_pac_out('pac_fpga', 'OUT7')
```

```
41 check_voltage('FPGA:SYS_2V5_24', lambda: read_pac_adc("pac_fpga", "VMON4"), 2.375, 2.625)
42 enable_pac_out('pac_fpga', 'OUT19')
43 enable_pac_out('pac_fpga', 'OUT18')
44 enable_pac_out('pac_fpga', 'OUT17')
45 enable_pac_out('pac_fpga', 'OUT10')
46 check_voltage('FPGA:MGTAVCC_FPGA', lambda: read_pac_adc("pac_fpga", "VMON7"), 0.855, 0.945)
47 enable_pac_out('pac_fpga', 'OUT14')
48 enable_pac_out('pac_fpga', 'OUT11')
49 check_voltage('FPGA:MGTVCCAUX_L', lambda: read_pac_adc("pac_fpga", "VMON8"), 1.71, 1.89)
50 enable_pac_out('pac_fpga', 'OUT12')
51 check_voltage('FPGA:MGTVCCAUX_R', lambda: read_pac_adc("pac_fpga", "VMON9"), 1.71, 1.89)
52 wait_for('B_CLOCK_CLOL', lambda: gpio.get_value('B_CLOCK_CLOL'), True, 10)
53 wait_for('B_CLOCK_FLOL', lambda: gpio.get_value('B_CLOCK_FLOL'), True, 10)
54 gpio.set_value('C_PLL_DCOK', True)
55 gpio.set_value('C_RESET_N', True)
```

Note: When creating the above listing, lines of code that were commented out have been removed.

A.2 Automatically generated sequence

Listing A.2: The automatically generated boot sequence

```
1 gpio.set_value('B_PSUP_ON', True)
2 check_voltage('3v3_psup', lambda: read_pac_adc('pac_cpu', 'VMON3'), 3.135, 3.465)
3 check_voltage('5v_psup', lambda: read_pac_adc('pac_fpga', 'VMON2'), 4.750, 5.250)
4 check_voltage('5v_psup', lambda: read_pac_adc('pac_cpu', 'VMON2'), 4.750, 5.250)
5 check_voltage('12v_cpu0_psup', lambda: read_pac_adc('pac_cpu', 'VMON1'), 4.702, 5.197)
6 check_voltage('12v_cpu1_psup', lambda: read_pac_adc('pac_fpga', 'VMON1'), 4.702, 5.197)
7 power.device_configure('clk_main', SI5395_Configuration.main)
8 power.device_configure('clk_cpu', SI5395_Configuration.cpu)
9 power.device_write('ir3581', 'address_lock', False)
10 power.device_write('ir3581', 'loop_1_pmbus_addr',96)
11 power.device_write('ir3581', 'loop_2_pmbus_addr',98)
12 power.device_write('ir3581', 'address_lock', True)
13 power.device_configure('ir3581', IR3581_Configuration.registers)
14 power.device_write('ir3581_loop_vdd_core', 'VOUT_COMMAND', 0.96)
15 enable_pac_out('pac_cpu', 'OUT6')
16 check_voltage('vdd_core', lambda: read_pac_adc('pac_cpu', 'VMON4'), 0.912, 1.008)
17 enable_pac_out('pac_cpu', 'OUT7')
18 enable_pac_out('pac_cpu', 'OUT8')
19 check_voltage('1v5_vdd_oct', lambda: read_pac_adc('pac_cpu', 'VMON6'), 1.425, 1.575)
20 power.device_write('ir3581_loop_0v9_vdd_oct', 'VOUT_COMMAND', 0.9)
21 check_voltage('0v9_vdd_oct', lambda: read_pac_adc('pac_cpu', 'VMON5'), 0.855, 0.945)
22 cdv_fdv_mmap[0] = chr(18)
23 time.sleep(0.05)
24 enable_pac_out('pac_cpu', 'OUT10')
25 enable_pac_out('pac_cpu', 'OUT9')
26 enable_pac_out('pac_cpu', 'OUT11')
27 enable_pac_out('pac_cpu', 'OUT12')
28 check_voltage('vtt_ddrcpu13', lambda: read_pac_adc('pac_cpu', 'VMON11'), 0.713, 0.787)
29 check_voltage('2v5_cpu13', lambda: read_pac_adc('pac_cpu', 'VMON7'), 2.375, 2.625)
30 check_voltage('vtt_ddrcpu24', lambda: read_pac_adc('pac_cpu', 'VMON12'), 0.713, 0.787)
31 check_voltage('2v5_cpu24', lambda: read_pac_adc('pac_cpu', 'VMON8'), 2.375, 2.625)
32 check_voltage('vdd_ddrcpu24', lambda: read_pac_adc('pac_cpu', 'VMON10'), 1.425, 1.575)
33 check_voltage('vdd_ddrcpu24', lambda : power.device_read('ina226_ddr_cpu_24', '
       BUS_VOLTAGE'), 1.425, 1.575)
34 check_voltage('vdd_ddrcpu13', lambda : power.device_read('ina226_ddr_cpu_13', '
       BUS_VOLTAGE'), 1.425, 1.575)
35 check_voltage('vdd_ddrcpu13', lambda: read_pac_adc('pac_cpu', 'VMON9'), 1.425, 1.575)
36 wait_for('B_CLOCK_BLOL', lambda: gpio.get_value('B_CLOCK_BLOL'), True, 10)
37 wait_for('B_CLOCK_CLOL', lambda: gpio.get_value('B_CLOCK_CLOL'), True, 10)
38 gpio.set_value('C_PLL_DCOK', True)
39 gpio.set_value('C_RESET_N', True)
40 power.device_configure('clk_fpga', SI5395_Configuration.fpga)
41 enable_pac_out('pac_fpga', 'OUT9')
42 enable_pac_out('pac_fpga', 'OUT6')
43 check_voltage('vccint_fpga', lambda: read_pac_adc('pac_fpga', 'VMON6'), 0.855, 0.945)
```

```
44 check_voltage('util33', lambda: read_pac_adc('pac_fpga', 'VMON3'), 3.135, 3.465)
45 enable_pac_out('pac_fpga', 'OUT13')
46 check_voltage('vccintio_bram_fpga', lambda: read_pac_adc('pac_fpga', 'VMON10'), 0.855,
       0.945)
47 enable_pac_out('pac_fpga', 'OUT15')
48 check_voltage('vcc1v8_fpga', lambda: read_pac_adc('pac_fpga', 'VMON11'), 1.710, 1.890)
49 enable_pac_out('pac_fpga', 'OUT8')
50 enable_pac_out('pac_fpga', 'OUT17')
51 cdv_fdv_mmap[8] = chr(66)
52 time.sleep(0.05)
53 enable_pac_out('pac_fpga', 'OUT7')
54 enable_pac_out('pac_fpga', 'OUT16')
55 enable_pac_out('pac_fpga', 'OUT18')
56 enable_pac_out('pac_fpga', 'OUT19')
57 check_voltage('sys_1v8', lambda: read_pac_adc('pac_fpga', 'VMON12'), 1.710, 1.890)
58 check_voltage('sys_2v5_24', lambda: read_pac_adc('pac_fpga', 'VMON5'), 2.375, 2.625)
59 check_voltage('sys_2v5_13', lambda: read_pac_adc('pac_fpga', 'VMON4'), 2.375, 2.625)
60 check_voltage('vdd_ddrfpga13', lambda : power.device_read('ina226_ddr_fpga_13', '
       BUS_VOLTAGE'), 1.140, 1.260)
61 check_voltage('vdd_ddrfpga24', lambda : power.device_read('ina226_ddr_fpga_24', '
       BUS_VOLTAGE'), 1.140, 1.260)
62 enable_pac_out('pac_fpga', 'OUT10')
63 check_voltage('mgtavcc_fpga', lambda: read_pac_adc('pac_fpga', 'VMON7'), 0.855, 0.945)
64 enable_pac_out('pac_fpga', 'OUT14')
65 enable_pac_out('pac_fpga', 'OUT12')
66 enable_pac_out('pac_fpga', 'OUT11')
67 check_voltage('mgtvccaux_l', lambda: read_pac_adc('pac_fpga', 'VMON8'), 1.710, 1.890)
68 check_voltage('mgtvccaux_r', lambda: read_pac_adc('pac_fpga', 'VMON9'), 1.710, 1.890)
69 wait_for('B_CLOCK_FLOL', lambda: gpio.get_value('B_CLOCK_FLOL'), True, 10)
```

A.3 Comparison

The table below summarises our comparison of the commands featured in the two boot sequences: The first two columns indicate the line numbers of the manually and automatically generated sequences that are *semantically equivalent* as defined in section 9.2. Where applicable, we have additionally included the designation of the corresponding conductor event.

Note that occasionally, a sequence of several commands $C1, C2, ...Cn$ might semantically correspond to a single other command C. In that case, we surrounded the line numbers of C1 to Cn with round brackets (). An example for this is the table row referencing line 15 of C_{man}: There, the manual sequence calls the helper function $program\_ir3581()$ that the automatically generated sequence inlines. In the summary of this table in section 9.2,

Furthermore, there occasionally exist multiple ways of monitoring the state of the same conductor, which results in different commands producing the same Complete event. In such cases, said event spans multiple rows, each of which is dedicated to one such monitoring command. Examples include table rows referencing lines 3 and 6 or 4 and 7 of C_{man}.

Lastly, when determining the syntactic equivalence of two commands, we have taken the liberty of ignoring the *identifier* passed to calls of the *check_voltage* functions. As already detailed in the evaluation sections, said identifier is only used for logging purposes and to generate meaningful error messages, and is hence not relevant in the context of this evaluation.

Line in C_{man}	**Line in C_{gen}**	**Syntactically Equal?** (Yes / reason why not)	**Corresponding Event**
1	1	helper function	Initiate(b_psup_on)
2	None (0)	-	-
3	5	margins	Complete(12v_cpu0_psup)
6	6	margins	
4	4	Yes	Complete(5v_psup)
7	3	Yes	
5	2	Yes	Complete(3v3_psup)
8	None (2)	-	-
9	None (2)	-	-
10	None (1)	-	-
11	None (1)	-	-
12	7	helper function	Initiate(clk_main)
13	8	helper function	Initiate(pll_ref_clk)
14	40	helper function	Initiate(fpga_clk)
15	(9 - 13)	helper function	(tied to vdd_core and 0v9_vdd_oct)
16	36	Yes	-
17	15	Yes	Initiate(vdd_core_en)
18	16	margins	Complete(vdd_core)
19	17	Yes	Initiate(vdd_oct_en_12)
20	21	margins	Complete(0v9_vdd_oct)
21	18	Yes	Initiate(en_1v5_vdd_oct)
22	19	Yes	Complete(1v5_vdd_oct)
23	25	Yes	Initiate(en_2v5_cpu13)
24	29	Yes	Complete(2v5_cpu13)
25	24	Yes	Initiate(en_2v5_cpu24)
26	31	Yes	Complete(2v5_cpu24)
27	26	Yes	Initiate(en_vdd_ddrcpu13)
28	27	Yes	Initiate(en_vdd_ddrcpu24)
29	42	Yes	Initiate(en_util33)
30	44	Yes	Complete(util33)
31	41	Yes	Initiate(en_vccint_fpga)
32	45	Yes	Initiate(en_vccintio_bram_fpga)
33	46	margins	Complete(vccintio_bram_fpga)
34	47	Yes	Initiate(en_vcc1v8_fpga)
35	48	Yes	Complete(vcc1v8_fpga)
36	54	Yes	Initiate(en_sys_1v8)
37	57	Yes	Complete(sys_1v8)
38	49	Yes	Initiate(en_sys_2v5_24)
39	58	Yes	Complete(sys_2v5_24)
40	53	Yes	Initiate(en_sys_2v5_13)
41	59	Yes	Complete(sys_2v5_13)
42	56	Yes	Initiate(en_vdd_ddrfpga24)
43	55	Yes	Initiate(en_vdd_ddrfpga13)
44	50	Yes	Initiate(en_vadj_1v8_fpga)

45	62	Yes	Initiate(en_mgtavcc_fpga)
46	63	Yes	Complete(mgtavcc_fpga)
47	64	Yes	Initiate(en_mgtavtt_fpga)
48	66	Yes	Initiate(en_mgtvccaux_l)
49	67	Yes	Complete(mgtvccaux_l)
50	65	Yes	Initiate(en_mgtvccaux_r)
51	68	Yes	Complete(mgtvccaux_r)
52	37	Yes	(tied to pll_dc_ok)
53	69	Yes	Initiate(clock_flol)
54	38	Yes	Initiate(pll_dc_ok)
55	39	Yes	Initiate(c_reset_n)
None (3)	14	-	Initiate(vdd_core)
None (3)	20	-	Initiate(0v9_vdd_oct)
None (4)	(22, 23)	-	Initiate(b_cdv_1v8)
None (4)	28	-	Complete(vtt_ddrcpu13)
None (4)	30	-	Complete(vtt_ddrcpu24)
None (4)	32	-	Complete(vdd_ddrcpu24)
None (4)	33	-	
None (4)	34	-	Complete(vdd_ddrcpu13)
None (4)	35	-	
None (4)	(51, 52)	-	Initiate(b_fdv_1v8)
None (4)	60	-	Complete(vdd_ddrfpga13)
None (4)	61	-	Complete(vdd_ddrfpga24)

Table A.1: A comparison of the commands found in the two boot sequences

We conclude this section by providing some explanations for the *None* instances in the table, which identify commands in either sequence that do not have a semantically corresponding command in the other sequence. For this purpose, we have labelled each such None instances with a number in brackets, which identifies the reason why no corresponding command exists:

(0) The single None instance identified with (0) is caused by the fact that we have previously not been aware of the existence of the PSUP_PGOOD signal, that indicates that the power supply's output is within regulation.

(1) The None instances marked with (1) are caused by commands that concern conductors that are not being modelled by our Enzian platform instance since they are not relevant from a power and clock management perspective.

(2) None instances of category (2) are caused by commands that establish the necessary boot-sequence preconditions. In the automatically generated sequence, these commands have already been created by the call to *apply_changes* that is performed prior to the consumer transitions.

(3) None instances marked with (3) are caused by redundant commands present in the automatically generated sequence: The IR3581 producer [18] on the Enzian platform requires a rather complex instantiation. As a part of this, the default voltages it should apply to its outputs as well as various other configuration parameters are written to the IR3581's registers. Regardless of this, our automatically generated procedure again specifies the voltages it has assigned to the IR3581's outputs, which in case of the boot-sequence happen to correspond to these default values and are therefore redundant.

(4) The manually generated sequence is not entirely complete: The handling of the DDR4 DRAM bank supply voltages of CPU and FPGA has not been included yet. This is the cause for the None instances marked with (4).

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

A model-based approach to power and clock management

Authored by (in block letters):

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

Name(s):	**First name(s):**
Schult	Jasmin

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)**
Rickenbach Sulz, 23.08.2020	J. Schult

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