μSystems Research Group

School of Engineering

Synthesis and Verification of Mixed-Signal Systems with Asynchronous Control

Vladimir Dubikhin

Technical Report Series

NCL-EEE-MICRO-TR-2020-217

November 2020

Contact: v.dubikhin1@ncl.ac.uk

Supported by EPSRC grant EP/L025507/1 and Dialog Semiconductor

NCL-EEE-MICRO-TR-2020-217

μSystems Research Group
School of Engineering
Merz Court
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my supervisor, Alex Yakovlev, for his wisdom and guidance throughout my research. He provided the invaluable opportunities to present my work as well as educated me to become a better researcher. I would also like to thank Danil Sokolov, who has helped me in developing the necessary skills in software design and publishing of scientific papers. I am very grateful to Chris Myers with whom I had the pleasure of working together all these years. He provided the critical feedback and helped me in shaping my ideas in numeral discussions.

I also extend my thanks to my colleges Jonathan Beaumont, Yuqing Xu, and Serhii Mileiko. I enjoyed our discussions and learned a lot about software and circuit design from you. I also want to express my gratitude to Thanasin Bunnam, who provided me the simulation data for the memristor example. Finally, I would like to thank David Lloyd and Carlos Calisto from Dialog Semiconductor for their help in implementation of the simulation testbench.

This research was supported by the EPSRC research grant 'A4A: Asynchronous design for Analogue electronics' (EP/L025507/1).

Publications

Conference papers

V. Dubikhin, C. Myers, D. Sokolov, I. Syranidis, and A. Yakovlev. Advances in formal methods for the design of analog/mixed-signal systems. In Proc. Design Automation Conference (DAC), 2017.

V. Dubikhin, D. Sokolov, C. J. Myers, A. Mokhov, and A. Yakovlev. Model discovery for analog/mixed-signal circuits. In FAC 2017; Frontiers in Analog CAD, pages 1–6, July 2017.

D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev. Benefits of asynchronous control for analog electronics: multiphase buck case study. In Proc. Design, Automation & Test in Europe (DATE), 2017.

Journal article

V. Dubikhin, D. Sokolov, A. Yakovlev, and C. J. Myers. Design of mixed-signal systems with asynchronous control. IEEE Design & Test, 33(5):44–55, 2016.

Abstract

Analog/mixed signal (AMS) systems are widely used in electronic devices, such as mobile phones, autonomous sensors, and radio transmitters. The traditional design flows are based on synchronous circuits, which simplify the design process but raise a number of limitations in certain applications. For example, in order to react promptly to the changes in an analog environment the control module needs to have a high clocking frequency. This in return leads to higher power consumption and wasted clock cycles, when no changes occur in the environment. Asynchronous circuits do not have this disadvantage as they react to input events at the rate they occur. However, with design automation being a huge concern asynchronous circuits are not widely used by industry.

Another problem related to the AMS system design is the reliance on simulation as the verification method. A simulation trace shows only one possible behavior of the system, as a result simulation based verification largely depends on quantity and diversity of tests. Formal methods, such as the reachability analysis, aim to address this problem. However, a lot of the proposed methodologies are disruptive to the existing design flows and require engineers to manually construct abstract models for their systems.

The main goal of this work is to introduce the novel automated workflow, which enables formal verification of AMS systems with asynchronous control that has been optimized with correct timing assumptions extracted from the full-system model. One of the key features of the proposed design flow is the ability to reuse existing simulation traces to generate abstract models, used for system validation. To overcome a number of flaws in the existing model generator a new version, utilizing data clusterization and process mining techniques, is created as a stand-alone framework in Java. The new model generator is designed to construct more general models that produce correct behavior, when used with a different control module.

Chapter 1. Introduction

From simple analog-to-digital and power converters to complex cellular network integrated circuits, *analog/mixed-signal* (AMS) systems are an essential component of many modern system-on-chip designs. Ever growing system complexity, performance and reliability requirements lead to an increased number of digitally assisted analog blocks [42]. This tight coupling of complex analog systems to complex digital solutions results not only in an extensive amount of verification to perform, but also necessity to combine design and validation methods for two fundamentally different system types.

1.1 Motivation

Existing trends in AMS system design, such as technology scaling, tight reliability margins, and short development cycle, put a great emphasis on design automation and verification [57]. While numerous tools have been developed for automation and verification of digital design, analog tool development has not kept pace. To cope with this problem, analog designers have turned to using digital alternatives whenever possible.

The design paradigm of digital components of AMS systems is different from the traditional computational electronics. As shown in Figure 1.1, such digital (on-top or within analog) elec-

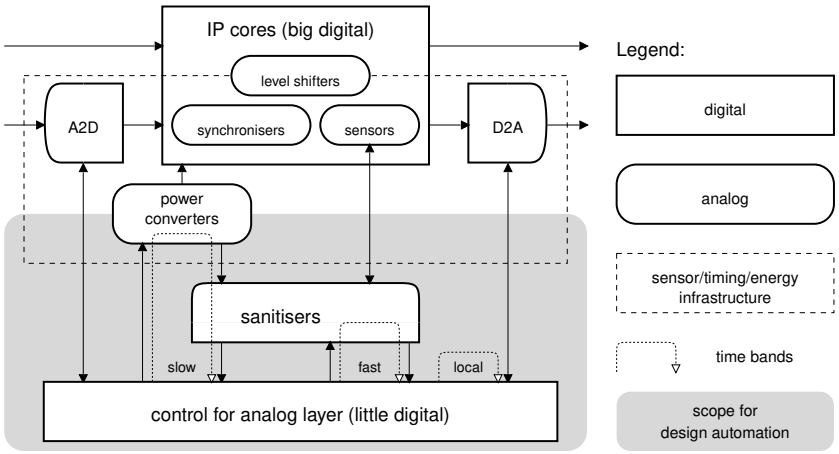


Figure 1.1: AMS system with “little digital” control

tronics are “little digital” as opposed to “big digital” electronics. Designing “little digital” is difficult because it should seamlessly integrate with the analog parts, which are dynamic and notoriously difficult to automate. Existing methods for digital design are based on synchronous circuits, which results in suboptimal solutions for mixed-signal systems [71]. The clocked operation mode, natural for the data processing, might lead to either low responsiveness or power consumption overheads in control modules of mixed-signal systems. On the one hand, the operating frequency must be sufficiently high to promptly react to changes in analog sensor readings. On the other hand, high clocking frequency can potentially result in wasted clock cycles if the sensors' readings change slowly.

For these reasons, the use of asynchronous logic for digital control has the potential to significantly improve the quality of the analog electronics. Asynchronous circuits can provide greater robustness, reactivity, and power efficiency. However, due to the lack of necessary computer-aided design tools, engineers have to rely on ad hoc development approaches and use extensive simulation to prove correctness of their designs. Furthermore, not only simulation time considerably increases with system's complexity, but also simulation driven verification is prone to human error and depends on the number and diversity of tests. This may result in longer development times and even the necessity to restart the whole project from the start due to some critical errors found at the final phase.

Furthermore, the fundamental problem in AMS system design is the gap in communication between analog and digital designers. Analog and digital domains evolved separately, which results in different tools and methodologies used to create analog and digital components. Additionally, analog design with “little digital” is largely done by analog engineers without any formal steps from the specification to netlists. No synthesis tools are in common use, and validation with conventional simulation can take days or longer. Formal design methods [22] can be used to improve the robustness of the designed systems as well as provide means to create abstract models of analog components. The latter is especially important as high level of abstraction for analog and mixed signal blocks can help to identify problems during early stages of development.

The main goal of this work is to improve the design process of AMS systems by adopting asynchronous circuit design and formal verification methods. While there is a number of such methods, as described in the following sections, there does not exist a unified methodology that would combine these areas. The novel design flow, detailed in Chapter 3, couples established asynchronous circuit design and formal verification tools and organically integrates into the existing simulation based design flow. An important aspect of the proposed design flow is the

usage of model generation, which allows to create abstract models from simulation traces and simplify verification process. Essentially, the model generation reuses existing simulation traces to automatically construct models of analog elements without requiring extensive knowledge of formal methods from engineers. As the model generation plays a key role in the design flow the major contribution of this work is an improvement over the existing model generation algorithm, as explained in Chapter 4. The improved model generation framework is capable of generating more general models of an analog environment, which can produce accurate behavior with new digital control module, as illustrated in Section 5.2 of Chapter 5.

1.2 Asynchronous circuit design

While early computers were either asynchronous, or had multiple asynchronous components, asynchronous circuits are no longer used as a mainstream implementation platform for electronic devices. The separation of functionality from timing, the necessary abstraction provided by synchronous logic, has paved way to the design of VLSI systems and as a result the modern digital implementation flow relies on synchronous circuits. However, the usage of the synchronization clock comes at a price, when moving to Deep Sub-Micron technology, as clock skew, power consumption and electromagnetic interference might become a problem. Asynchronous approach offers a number of significant advantages over traditional design methods [18]:

- **Modularity.** The ability to *plug and play* existing designs, without having to re-do the clock routing and fabrication, can reduce the development time.

- **Power Consumption and Electromagnetic Interference.** Asynchronous control circuits tend to be quiet, as they avoid unneeded transitions, due to absence of the clock, while spreading out needed transitions.

- **Performance.** Not only asynchronous modules may exhibit average case performance, rather than worst case as synchronous ones, but also can be finely tuned down to the levels of transistor sizing and individual transition delay, which can in turn improve overall system performance.

Although, good on paper the design of asynchronous circuits is more difficult than synchronous ones and is largely done by hand. In order to tackle the problem of asynchronous design automation a number of design methodologies has been developed [25, 55]. These tools, utilizing

various delay models from bundled delay to quasi-delay-insensitive circuits, allow for automated asynchronous system specification and synthesis.

1.2.1 *ATACS*

ATACS [2] is a synthesis, analysis, and verification tool for timed circuits. The tool accepts designs given in various formats including VHDL, Petri nets, timed event/level structures, and state graphs. Analysis is performed by running a stochastic simulation utilizing provided delay information. For verification, a set of timing constraints is checked during timed state space exploration. Afterwards, verified design can be synthesized via one of synthesis algorithms, such as *binary decision diagrams* (BDD) or direct synthesis. Each synthesis method generates circuits which are hazard-free under a particular technology model.

The tool has been used in the verification of delayed-reset domino circuits in the guTS (gigahertz unit Test Site) processor [12]. A delayed-reset domino macro consists of a number of levels of dynamic gates, each of which receives inputs from preceding layers. Formal verification assured design correctness and gave confidence that all of the timing behaviors have been considered.

1.2.2 *Petrify*

PETRIFY [4] is a tool for the synthesis of Petri nets and asynchronous controllers [17]. Given a *Petri Net* (PN), a *Signal Transition Graph* (STG), or a *Transition System* (TS) it can generate another PN or STG, which is simpler than the original description, and produce an optimized net-list of an asynchronous controller in the target gate library, while preserving the specified input-output behavior.

For synthesis of an asynchronous circuit petrify performs state assignment by solving the Complete State Coding problem. State assignment is coupled with logic minimization and speed-independent technology mapping to a target library. The final net-list is guaranteed to be speed-independent, i.e., hazard-free under any distribution of gate delays and multiple input changes satisfying the initial specification. The tool has been used for synthesis of PNs and PNs composition, synthesis and re-synthesis of asynchronous controllers and can be also applied in areas related with the analysis of concurrent programs.

1.2.3 *MPSAT*

MPSAT [35] provides an extensive set of functions for composition and verification of Petri nets and STGs, and for synthesis of electronic circuits from STGs. Additionally, to cope with the problems of expressing verification properties for PNs the developers have designed a new property expression language REACH to allow easy and concise specification of custom properties. The tool builds a finite and complete prefix of the given Petri net unfolding to represent a PNs state space. For STGs such a representation is often superior to that based on explicit state graphs or BDDs due to the fact that STGs usually contain a lot of concurrency but rather few choices. As a result, the memory requirements of synthesis algorithms based on unfoldings are very moderate.

1.3 AMS formal verification

As an alternative to simulation-based verification, numerous researchers have been exploring the application of formal verification methods to AMS circuits. Formal verification utilizes exhaustive algorithmic techniques to ensure that a design implementation satisfies the properties given in its specification [56]. These properties are often expressed using temporal logic, while the model for the design can be expressed formally in a variety of ways including automata, Petri nets, etc. Formal verification then proceeds to exhaustively check that the properties are satisfied. In the end, if the formal representation of the system is correct and the set of properties precisely characterize the specification requirements, then the designer can have a higher confidence of correct operation.

The key challenge with the application of digital domain formal methods to the analog domain is the continuous nature of voltage and current state variables. Therefore, formal approaches in the AMS space must deal with a potentially infinite state space. To cope with this problem researches have proposed a number of various methods, described below.

1.3.1 *Theorem Proving*

METITARSKI is an automatic theorem prover based on a combination of resolution and a decision procedure for the theory of real closed fields. It is designed to prove theorems involving real-valued special functions such as log, exp, sin, cos and sqrt. In particular, it is designed to prove universally quantified inequalities involving such functions. METITARSKI has been lately used to determine the possibility of oscillation of a tunnel diode oscillator and the change in gain due to component tolerances for an operational amplifier [21]. In [53] Narayanan et al. adopted

the METITARSKI tool-set to verify saturation property of an Op-Amp under noise and process variation conditions.

1.3.2 *DC Operating Point Analysis*

These approaches assume the inputs are held steady and try to find a unique equilibrium point. One such approach is implemented in the FSPICE tool, which solves the multiple DC operating points problem by setting up and solving a *satisfiability* (SAT) problem [70]. Other techniques for DC analysis can be found in [32, 76, 69], and [78]. The latter, for example, applies evolutionary computing for the detection of multiple equilibrium points.

1.3.3 *Equivalence Checking*

These approaches attempt to show that two representations of an AMS circuit produce the same response to the same inputs. In [42], a new flow is proposed to enable a top-down design approach for analog components. Analog cells are described using SystemVerilog and compared against their implementation at the transistor level, while digital blocks are validated using existing tools for digital components. This validation method has been used to test analog cells of a single-slope ADC and a serial link receiver.

1.3.4 *Symbolic Simulation*

Authors of [60] present an extension to the symbolic simulation approach utilizing affine arithmetic to allow the representation of control flow and discrete changes. The proposed methodology is used to verify the stability property of a 3rd order $\Sigma\Delta$ modulator.

Another approach using affine arithmetic, described in [31], tackles the problems of device mismatch and process variation. They reformulate the basic *modified nodal analysis* (MNA) equations in order to include vectors containing parameter expressions based on affine arithmetic. The result of the simulation is not a single trace but a range of traces capturing all potential simulation results obtained by varying a parameter in a certain range. The methodology is applied to an analog band-pass filter and the results are compared to a Monte Carlo simulation. The simulation takes less time, however the algorithm tends to diverge when strong nonlinearities occur.

1.3.5 *State Space Guided Simulation*

In [68], the authors propose a property verification and equivalence checking methodology for analog circuit blocks based on a novel algorithm for formal automatic input stimuli generation. Therewith, it overcomes the incompleteness of transient simulation and the designer-unfriendliness of formal approaches by combining a formal approach and conventional transient circuit simulation. This method is applied to a Sallen-Key bi-quad low-pass filter with a cut-off frequency of 1000 Hz. Using a property specification for overshoot behavior and an automatic evaluation on the simulation result, complete and, therefore, formal property verification coverage is obtained without user-interaction.

1.3.6 *Reachability Analysis*

The COHO tool performs reachability analysis using state spaces represented as projections of high-dimensional polyhedra onto high-dimensional spaces [74, 75]. This method is successfully used to verify the correctness of a high speed toggle element and an arbiter. Verification of cyclic properties can also be performed by proving the existence of a cyclic invariant.

The PHAVER tool [29] operates on *linear hybrid automata* (LHA) which by definition contain both discrete and continuous components. Similar to differential inclusions, LHA are characterized by a set of states and linear inequalities defining transitions. For the computation of the reachable states, PHAVER uses a polyhedral representation and over-approximation based on affine dynamics.

The SPACEEX tool [30] provides an extensible verification platform for hybrid systems. The tool consists of three main components: an analysis core, a command line program, that analyses the system; a web interface, which provides the ability to specify initial states and other analysis parameters, run the analysis core, and visualize the output graphically; and a model editor, a graphical editor for creating models of complex hybrid systems out of nested components. SPACEEX relies on hybrid automata for model description and support functions [41] for state space exploration. This system has been used to model and verify the behavior of several benchmarks [54, 49].

Another work [7] presents a state space analysis method for verifying both the transient and invariant specifications for a PLL using zonotopes by describing reachable sets. The behavioral model of the charge-pump PLL is a hybrid automaton with linear continuous dynamics and uncertain parameters. Furthermore, authors claim that their methodology computes accurate over-approximations of reachable sets for hybrid systems when there are a large number of

discrete state transitions. The methodology is applied to the verification of locking time and stability of a 27GHz PLL designed in 32nm CMOS SOI technology. The novel reachability analysis method efficiently provides an upper bound on the worst-case lock time in the presence of random phase error and charge pump current variations.

1.4 Contributions

This dissertation presents a number of improvements in the design of AMS systems, featuring usage of asynchronous logic for digital control and formal verification methods of the entire AMS design. The main contributions of this work are summarized below:

- Analysis of the multiphase buck converter with asynchronous control.
- Development of the new AMS design flow, based on tools WORKCRAFT and LEMA.
- Design of the singlephase buck converter with asynchronous control and control optimization.
- Design and implementation of the new methodology for generating AMS models from simulation traces.

The first contribution provides comparative analysis of the existing asynchronous control module vs its synchronous counterpart. An AMS simulation testbench, written in Verilog-A, has been built to obtain a number of metrics from mixed-signal simulation. Asynchronous circuit has proven to have faster reaction time and potential for coil reduction, while maintaining the same current ripple level.

The second contribution focuses on bringing together two research tools in a joint AMS design flow. WORKCRAFT [5] allows automated synthesis of asynchronous circuits from their STG specification, while LEMA [3] provides means to construct abstract models of the entire AMS system and perform formal verification through reachability analysis. The core idea of the new design flow is to reuse existing simulation traces and convert them into an abstract model of the analog environment. This model in combination with the STG specification can not only help increase reliability of AMS circuits, but also allow exploitation of timing information, extracted from simulation.

The third contribution applies the proposed design flow to improve the design of a singlephase buck converter. The buck converter with asynchronous control is used as a motivational example to test the new flow and identify any underlying problems. Using the existing simulation testbench several simulation traces have been obtained to generate abstract models of

analog blocks. These models in combination with the STG specification have been used to formally verify operation of the design. At the same time a new optimization algorithm, which operates on the produced state graph of the system, has been capable to identify possibilities for control optimization, such as concurrency reduction and scenario elimination.

And, finally, the fourth contribution is the implementation of the new model generation algorithm. Several critical issues with the existing model generation module in LEMA have been identified, which can limit quality and precision of the generated models. To overcome these problems a new model generation flow has been developed. The proposed methodology aims to improve model precision by introducing new derivative based discretization method. Furthermore, a fine control over model fitness is provided via a notion of data rule mining, a novel approach to finding recurring patterns in the input data.

1.5 Dissertation Overview

The rest of the dissertation is divided into five chapters:

- Chapter 2 expands on asynchronous control design principles, introduced in previous chapter. The chapter describes the tool WORKCRAFT and provides the necessary background information, regarding STG semantics. Later on, the chapter explains the operation of the new multiphase buck converter and proves that asynchronous control is superior to synchronous one.

- Chapter 3 introduces the tool LEMA and the new LEMA-Workcraft AMS design flow. Single-phase buck converter is to be used as motivational example to demonstrate that the new flow can not only formally verify the operation of the system, but also provide means for automatically identifying possibilities for control optimization.

- Chapter 4 highlights existing problems withing the model generation module in LEMA and offers new methodology for generating abstract models from simulation traces. The proposed methodology seeks to provide a finer control over ranges of rates as well as an overall model structure.

- Chapter 5 provides further explanation to the operation of the new model generator by applying it to a number of digital and analog components, such as a memristor.

- Chapter 6 summarizes the contributions of this dissertation. It recaps the advantages of asynchronous circuits and how the new flow can improve the design of AMS systems.

The chapter concludes with a list of possible improvements to the new model generation approach.

Chapter 2. AMS system design

Analog and mixed signal electronics govern distribution and regulation of energy flows, monitoring of the system's operating conditions, and interfacing with the analog environment. Power converters [59] are of particular importance as energy is becoming the most valuable resource in modern electronics. The responsiveness and robustness of power converters is determined by the implementation of their control circuitry: millions of control decisions need to be made every second and a single incorrect decision may cause a malfunction of the whole system or even permanently damage the circuit. For example, a 3MHz switching regulator is clocked around 473,364,000,000,000 times in 5 years of its operation [8].

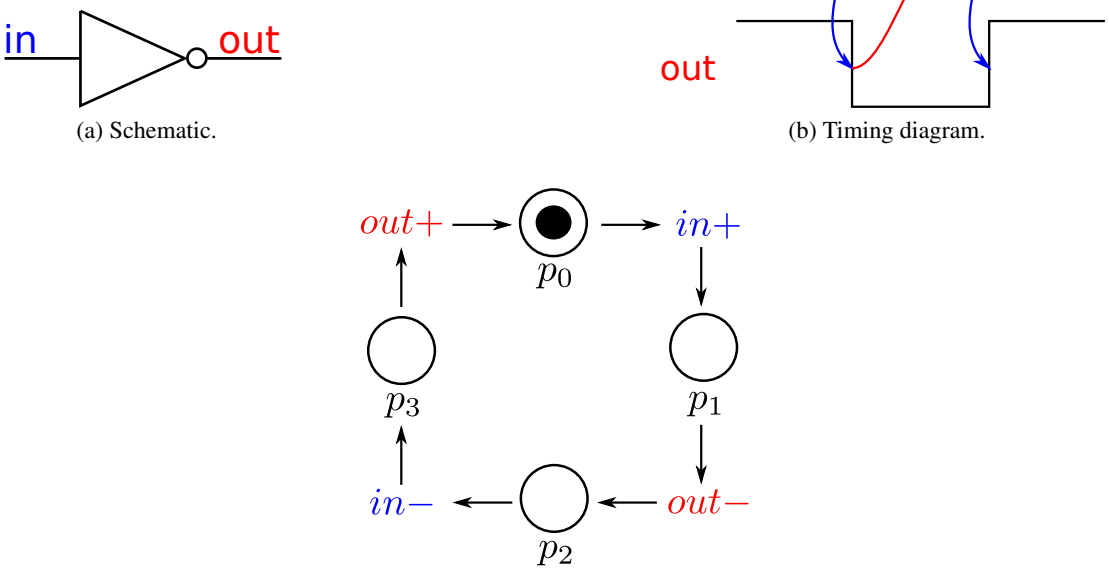
AMS control can significantly benefit from the use of asynchronous logic [72] that does not rely on the global clocking and operates at the pace determined by the operating conditions. Hence, there is an ever increasing scope for the use of *asynchronous design for analog electronics* (A4A). There are many methodologies and design styles for asynchronous circuits [67], however few of them tackle the problem of design automation. *Ad hoc* solutions, based on unrealistic assumptions, may lead to hazards propagating into the digital core of the system.

This chapter introduces the new A4A design flow, based on the STG formalism and supported by WORCKRAFT framework. This flow, developed by the asynchronous community from the Newcastle university, is applied to the design of the control module for a multiphase buck converter to demonstrate benefits of asynchronous controllers over the traditional synchronous ones.

2.1 Signal Transition Graphs

Signal transition graph, a special type of a Petri net, provides excellent capabilities for capturing concurrent behavior of asynchronous circuits as well as a necessary pragmatic design notation [18]. An STG describes causality relations between input and output signals of a system by associating rising and falling edges of a signal with a Petri net transitions. Formally, an STG can be defined as a tuple $S = \langle P, T, F, M_0, \lambda, Z, v \rangle$ where:

- P is a finite set of places;



(a) Schematic.

(b) Timing diagram.

(c) Signal transition graph.

Figure 2.1: Inverter example.

- T is a finite set of transitions;
- $F \subseteq (P \times T) \bigcup (T \times P)$ is the flow relation;
- $M_0 \subseteq P$ is the set of initially marked places;
- λ is a labeling function;
- Z is a set of signals;
- v is a vector of initial signal values.

The operation of an STG is best illustrated with the following examples. Consider a digital inverter, as shown in Figure 2.1a. The traditional timing diagram in Figure 2.1b describes timing and causal relations between signals *in* and *out*. These relations, expressed as colored arcs, define the order in which the signals can appear in a simulation trace. For example, the positive edge of the signal *in* causes the signal *out* to change its value from 1 to 0. Similarly, a negative edge of signal *in* can only occur after the signal *out* transitioned to 0. The STG specification of this system, provided in Figure 2.1c, captures this behavior in the form of a Petri net.

The values of the signals *in* and *out* are assumed to be 0 and 1, respectively. Initially, transition *in*+ is enabled, which corresponds to the positive edge of the *in* signal. Firing this transition changes the value of this signal to 1 and enables the transition *out*-, which changes the value of the *out* signal to 0. This effectively represents the causal relation between these signals, expressed by the first red and blue arcs on the timing diagram. In a similar manner the relation between the *in* negative edge and *out* positive edge is formulated.

In order for an STG model to be used for asynchronous circuit design it must possess a few important properties. First of all the STG must be one place safe, which means that no more than one token is permitted in any place at any time. The STG model must also be consistent, which states that for any signal sequences labeled with + or - alternate in any firing sequence. Additionally, the model must be persistent, indicating that no transition can be disabled by another transition unless they both are events of different input signals. Finally, the model must be free of deadlocks. It is important to note that for simplicity reason places are often omitted from a graphical representation of an STG model.

2.2 Workcraft

WORKCRAFT is a tool-set for capture, simulation, synthesis, and verification of interpreted graph models. The tool provides convenient mechanism for verification of constructed STGs and subsequent high-level synthesis, using one of back-end tools: PETRIFY [4] or MPSAT [35]. The main features of this framework are [66]:

- *Availability* - open-source front-end and plugins, permissive licenses for back-end tools, as well as frequent releases with bug fixes and features requested by users.
- *Usability* - elaborated GUI that was developed with much feedback from the users.
- *Portability* - it runs on Windows, Linux, and Mac OS X operating systems.
- *Extendibility* - the framework is designed to easily include new interfaces to back-end tools as plugins.
- *Automation* - several complete design flows have been implemented by bridging the gaps between back-ends and converting file formats.

WORKCRAFT is designed to provide a common framework for the development of *Interpreted Graph Models* (IGMs), such as Finite State Machines, Petri nets, Signal Transition Graphs,

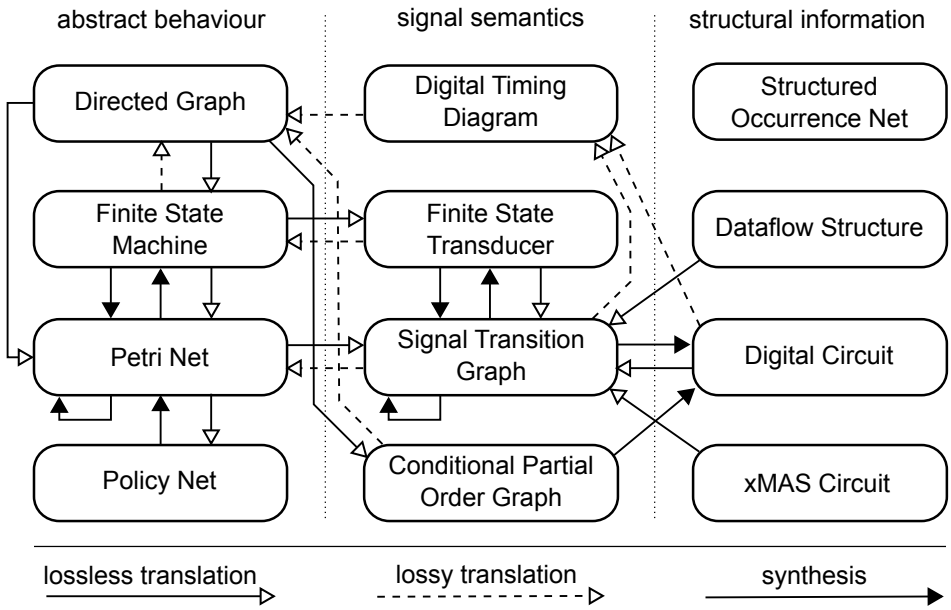


Figure 2.2: Model relationship.

gate-level circuits, and dataflow structures. Consequently, the tool can be used for a wide variety of applications ranging from modeling concurrent algorithms and biological systems to designing asynchronous circuits and investigating crimes. WORKCRAFT allows to automatically convert from one model into another formalism, as presented in Figure 2.2. For example, a verilog netlist can be converted into an STG format, which can be transformed into a finite state transducer.

While the primary goal of the tool is the automated synthesis of asynchronous circuits [65], this flexibility allows WORKCRAFT to be used in multiple applications. For example, in [19] the tool was used to specify and explore *Instruction Set Architecture* (ISA) of the ARM CORTEX-M0 processor as well as synthesize an efficient processor microcontroller. Thanks to the dataflow structure (DFS) support the tool proved useful in designing and verifying an asynchronous dataflow accelerator for reconfigurable ordinal pattern encoding [62]. Finally, the plugin based architecture allows to incorporate new asynchronous design methodologies, such the synthesis of an STG specification from a number of high-level concepts [11].

2.3 A4A design flow

The A4A design flow, as shown in Figure 2.3, starts with an informal specification of the intended system behavior in the form of phase diagrams, waveforms, and verbal requests [63]. This needs to be formalized in a consistent and unambiguous form of signal transition graphs.

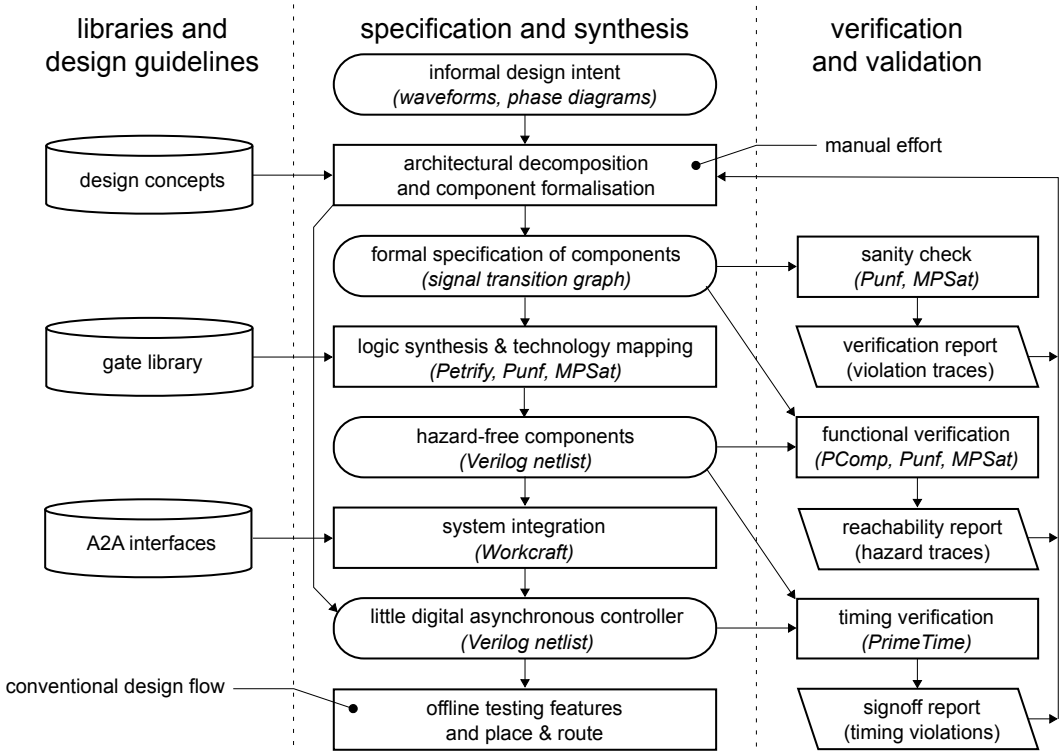


Figure 2.3: A4A design flow.

The STGs are either manually created using GUI or imported from a ***.g** file, produced by PETRIFY. This step, being difficult to automate, relies on the designer experience and the established design guidelines for decomposition of the design into simple sub-modules.

Afterwards, the designer verifies various properties of the specification, such as consistency or deadlock freeness, while the tool helps the user to debug the STG by visualizing reports from a back-end tool. The verified specifications of the sub-modules are synthesized into speed-independent gate-level components [16], which are integrated into a complete "little digital" controller. The circuit can be further verified or simulated to ensure no errors occurred during the synthesis. Standard *electronic design automation* (EDA) tools can be reused for place-and-route and offline testing of asynchronous "little digital" controllers [46]. For interacting with the analog components the asynchronous controller relies on a library of *analog-to-asynchronous* (A2A) [65, 64] interface elements for sanitizing non-persistent inputs. The next sections illustrate how the proposed methodology can be applied to the design of a multiphase buck converter.

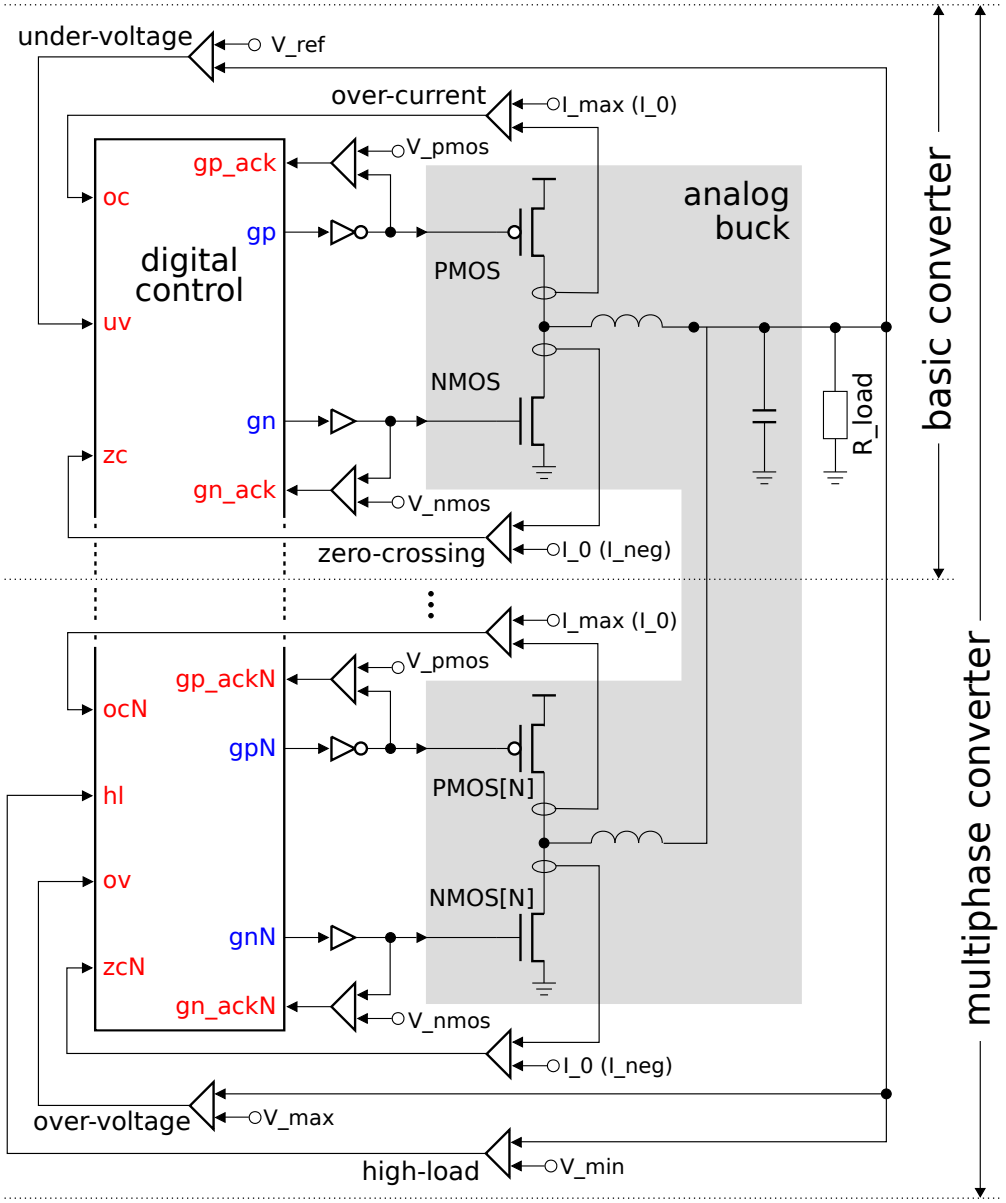
2.4 Multiphase buck

A buck converter comprises an analog buck and a digital control [59], as shown in Figure 2.4a. A *basic* buck converter consists of the digital control, which determines the state of the PMOS and NMOS transistors, depending on the input sensors. These transistors are switched on and off periodically to step-down the input voltage and power the load. The inductor and capacitor act as an **LC** filter to smooth the output voltage and provide steady voltage level for the load.

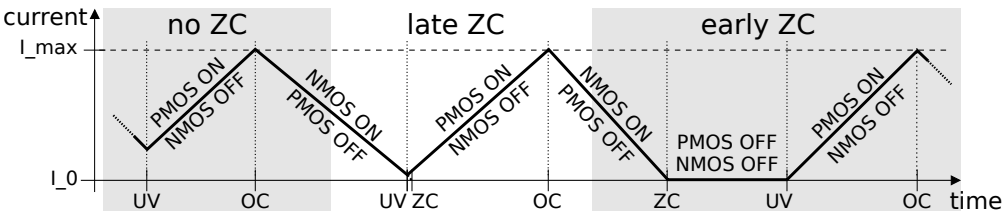
The state of the power transistors is determined by three main conditions: *under-voltage* (UV), *over-current* (OC) and *zero-crossing* (ZC) conditions (uv, oc and zc inputs respectively). These conditions are detected by a set of sensors that compare the measured current and voltage with some reference values (V_ref, I_max, I_0). *Under-voltage* shows that the output voltage is below a specified minimum and a charging cycle must start by switching the NMOS transistor OFF and PMOS transistor ON, while the *over-current* signal reports that the current flowing through the system has reached the maximum safe limit and thus the state of the transistors must be reverted - PMOS OFF and NMOS ON. Additionally, *zero-crossing* signal indicates the direction of the current flowing through the coil and control prevents the reversal of the current flow by turning both transistors off. Note that in order to avoid a short-circuit the PMOS and NMOS transistors of the buck must never be ON at the same time. Therefore, the controller is explicitly notified (by the gp_ack and gn_ack signals) when the power transistor threshold levels (V_pmos and V_nmos) are crossed.

The operation of buck converter is usually specified in a rather informal way, for example by describing sequences of input conditions and the intended reactions to these evens, as shown in Figure 2.4b. This diagram shows that the UV condition is handled by turning the PMOS transistor ON and the NMOS transistor OFF, while the OC condition reverts the state of these transistors to PMOS OFF and NMOS ON. Detection of the ZC condition after UV does not change this behavior. However, if ZC is detected before UV then both transistors must be turned off until the UV condition.

A *multiphase* buck converter [9] combines several pairs of PMOS and NMOS transistors, called *phases*, to power the same load, as illustrated in Figure 2.4a. The main advantages of this distributed design compared to the basic buck are faster reaction to the power demand, heat dissipation from a larger area, and the possibility to replace a large coil with several smaller ones, thus reducing the dimensions of consumer gadgets [59]. The control circuit of a multiphase buck monitors over-current and zero-crossing conditions of individual phases and the voltage



(a) Schematic.



(b) Informal specification of a basic buck.

Figure 2.4: Buck converter.

level at the load. When the UV condition is detected the controller performs a charging cycle at the currently active phase. Traditionally, the active phase is selected in a round-robin manner and only one phase can be active under normal conditions.

The high-load (HL) condition indicates a sudden increase in power demand, which corresponds to the voltage drop below V_min[1] value. As a result the controller reacts by starting a charging cycle in all phases simultaneously. As buck rumps to its target voltage it can overshoot, which leads to an excessive voltage on the capacitor and can damage the connected system. To mitigate this problem the controller enters the over-voltage (OV) mode to sink extra energy. This is achieved by changing reference values for OC and ZC conditions, so that PMOS is switched OFF as soon as positive current is detected and NMOS stays ON until the negative current limit is reached.

Note that for efficiency reasons, once ON, the PMOS and NMOS transistors should not switch OFF for at least the predefined PMIN and NMIN time intervals, respectively. Furthermore, at the first charging cycle after the arrival of UV the PMOS transistor should stay ON even longer (PMIN + PEXT).

2.4.1 *Synchronous control*

A high-level architecture of the synchronous multiphase buck controller is presented in Figure 2.5. The module consists of N phase control modules and a phase activator that selects the phase controllers in a round-robin pattern. The signals, coming from the environment, are sanitized by a set of synchronizers [36] (shaded components), implemented as a pair of flip-flops. The processed input signals are used by other modules that a specified in conventional RTL style as a clocked finite state machine (FSM) and synthesized by the standard EDA flow.

There are two clocks in this design: relatively slow clock for generating non-overlapping pulses for phase activation, and a fast clock for reading sensors and clocking the FSM. The latter clock is implicitly connected to synchronizers and other system components. Note that synchronization imposes a latency of up to 2.5 clock periods in the reaction time of the synchronous buck controller. Two clocks due to usage of double flip-flop synchronizers and 0.5 clocks for FSM operation. The latter delay is achieved by doing the synchronization on the negative clock edge and the FSM computation on the positive one. This latency can be increased if a synchronizer hits metastability.

[1] It is implied that V_min < V_ref.

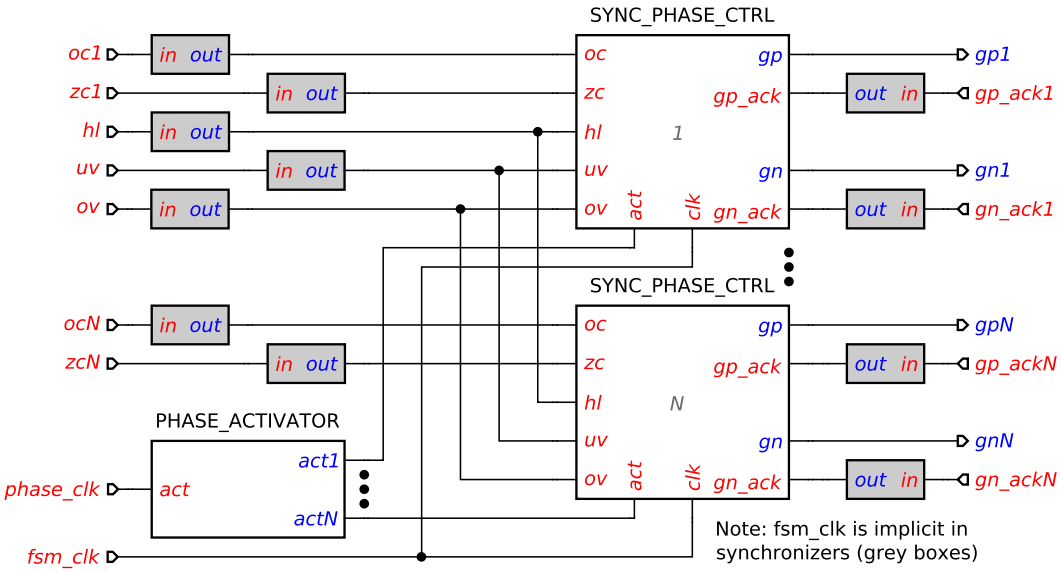


Figure 2.5: Synchronous control module.

2.4.2 *Asynchronous control*

Asynchronous control for the multiphase buck does not have an explicit phase activator and instead utilizes a token ring architecture for phase activation [64], as shown in Figure 2.6. On receiving a token a phase becomes active and starts a charging cycle. The token is passed to the next connected stage after a predefined duration of time. A single phase controller, illustrated in Figure 2.7, consists of several sub-modules, responsible for token propagation, synchronization of input signals with internal logic, and performing a charge cycle. The asynchronous sub-modules communicate by means of handshakes with the following naming convention: requests start with 'r' and acknowledgements with 'a'.

A number of A2A interface components (shaded) are used to process and sanitize input signals. In contrast to the conventional synchronizers these components are event driven, which means that instead of continuously polling its state they wait for the specific change of a non-persistent input. For example, a **WAIT** module is used to latch the HL condition and is analogous to a synchronizer. A **WAIT01** is similar in operation, however it detects a rising edge of the input signal rather than being triggered only by an input value. A **WAITX2** component identifies the UV and OV modes and performs arbitration between them. An **RWAIT** element, which is used to wait for ZC condition, is a modification of the **WAIT** with an added possibility to persistently cancel the waiting request. Finally, a **WAIT2** module, which monitors the state of the OC condition, uses a 2-phase output handshake, waiting for high and low input values, one after the other.

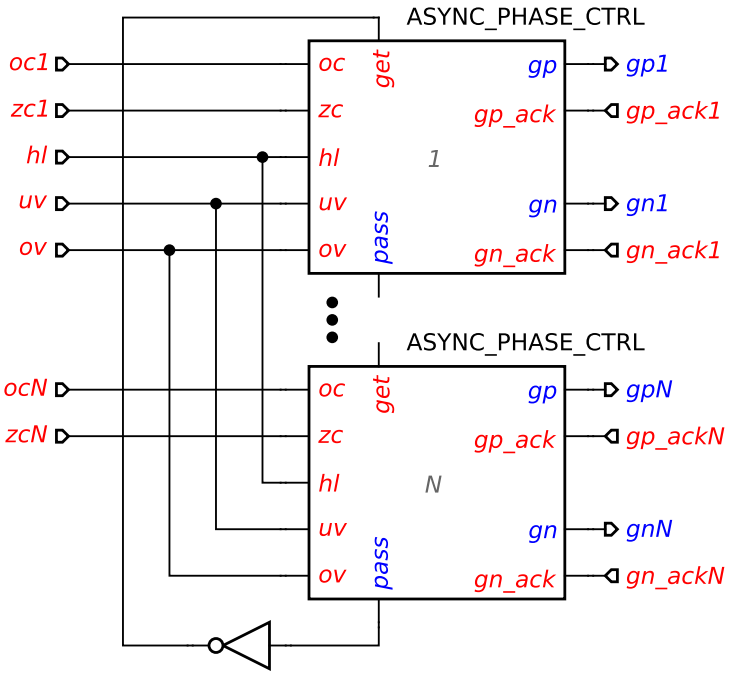


Figure 2.6: Asynchronous control module.

The phase controller has two distinct functions, handling its activation and charging the buck. The stage becomes active on receiving a token from the previous stage via the **get/-pass** interface or when when the HL condition is detected by the **HL_CTRL** module. The OR-causality between these scenarios is handled by the MERGE component which is implemented using the *opportunistic merge* element [52]. The **TOKEN_CTRL** module activates **TOKEN_TIMER** to delay passing the token to the next phase and at the same time activates the **MODE_CTRL** component, which monitors OV and UV conditions and determines the mode of operation. **CHARGE_CTRL** executes a charging cycle similar to that of the basic buck. The **DELAY_CTRL** modules ensure the minimum ON time for PMOS and NMOS transistors by delaying the corresponding acknowledgements. They employ **PMIN_TIMER** and **NMIN_TIMER**, respectively, to specify the delays. Furthermore, **PEXT_TIMER** is used to keep PMOS transistor ON longer on the first cycle of charging in the UV mode.

STG specifications of all controller modules were developed, synthesized and verified using WORKCRAFT framework [58, 5]. The produced STGs were verified to ensure consistency, deadlock-freeness, and output-persistency. Additionally, specific buck converter properties, such as the absence of a short circuit in PMOS/NMOS transistors, were verified. All the gate-level implementations were also verified to be deadlock-free, hazard-free and conformant to their STG specifications.

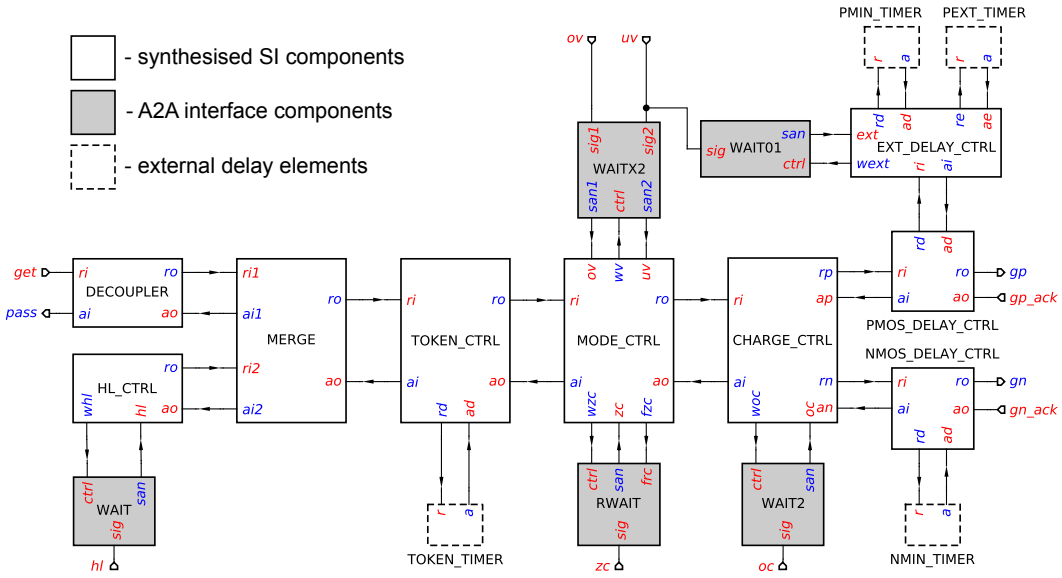


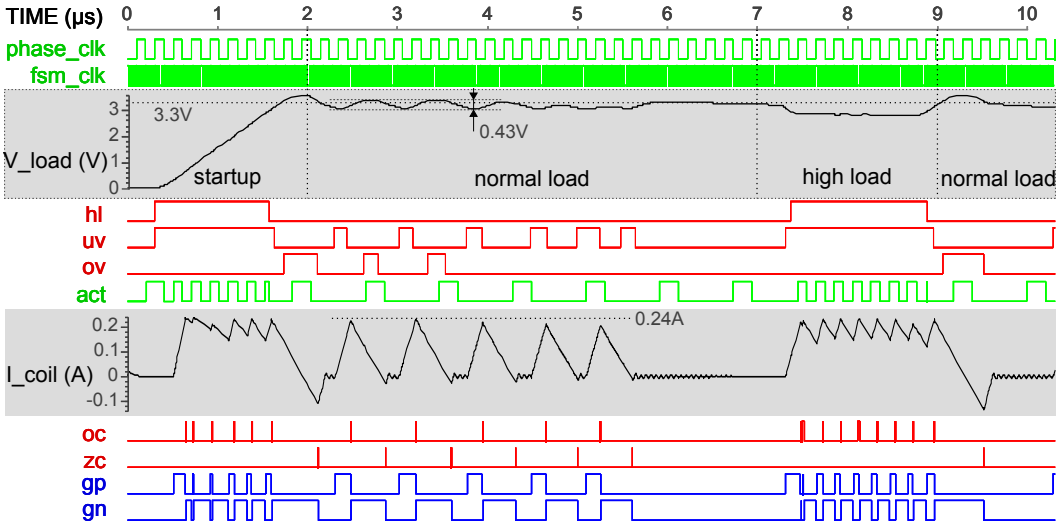
Figure 2.7: Asynchronous phase controller.

2.5 Results and analysis

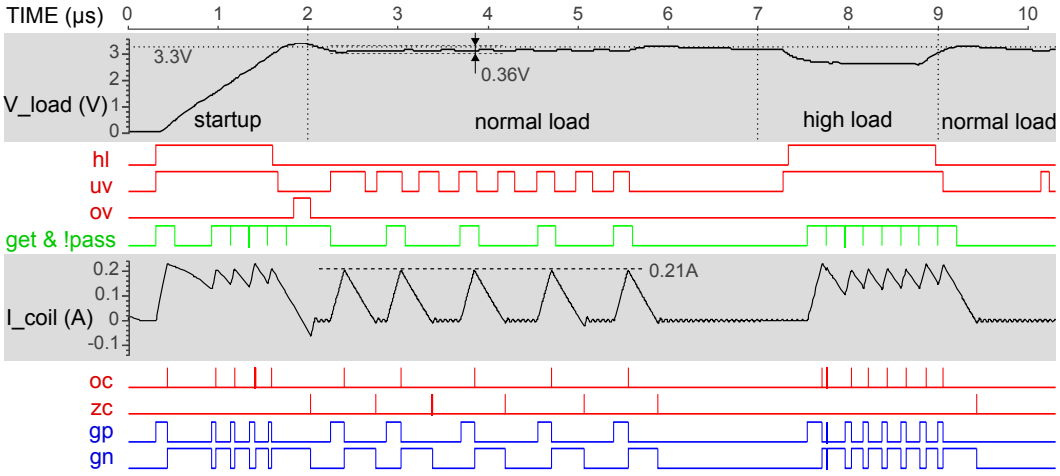
A 4-phase buck using synchronous and asynchronous controllers was implemented. The analog components were modeled in Verilog-A and the digital control was implemented in TSMC 90nm technology. Synchronous controller was synthesized using Synopsys Design Compiler for 100MHz, 333MHz, 666MHz, and 1GHz. Response time of synchronous control is 2.5 clock periods. The latency of asynchronous design was measured in Synopsys PrimeTime.

The operation of the buck was validated and its efficiency was estimated by simulation with Cadence Incisive using an AMS testbench. Coils were modeled in the range from 1μH to 10μH using the parameters of Coilcraft RF inductors [1]. Figure 2.8 shows the simulation waveforms for one of the buck phases. One can notice that the asynchronous buck enjoys smaller voltage overshoot after resolving the first HL condition at buck *startup* (1-2μs). This results in shorter OV resolution time and absence of recurring OV conditions, which can be observed in the synchronous buck waveform (2-4μs). Note that asynchronous buck does not even overshoot at the exit from *high load* (7-8μs). At *normal load* (2-7μs) asynchronous buck demonstrates smaller voltage ripple and lower inductor peak current than the synchronous buck: 0.36V vs 0.43V and 0.21A vs 0.24A, respectively. These advantages are due to faster reaction of the asynchronous controller to the input stimuli, as summarized in Table 2.1. To achieve response times similar to the asynchronous controller, the synchronous design would need to be clocked at 3GHz, which is impractical.

The quick response of the asynchronous control enables it to operate with a significantly



(a) Synchronous control at 333MHz.

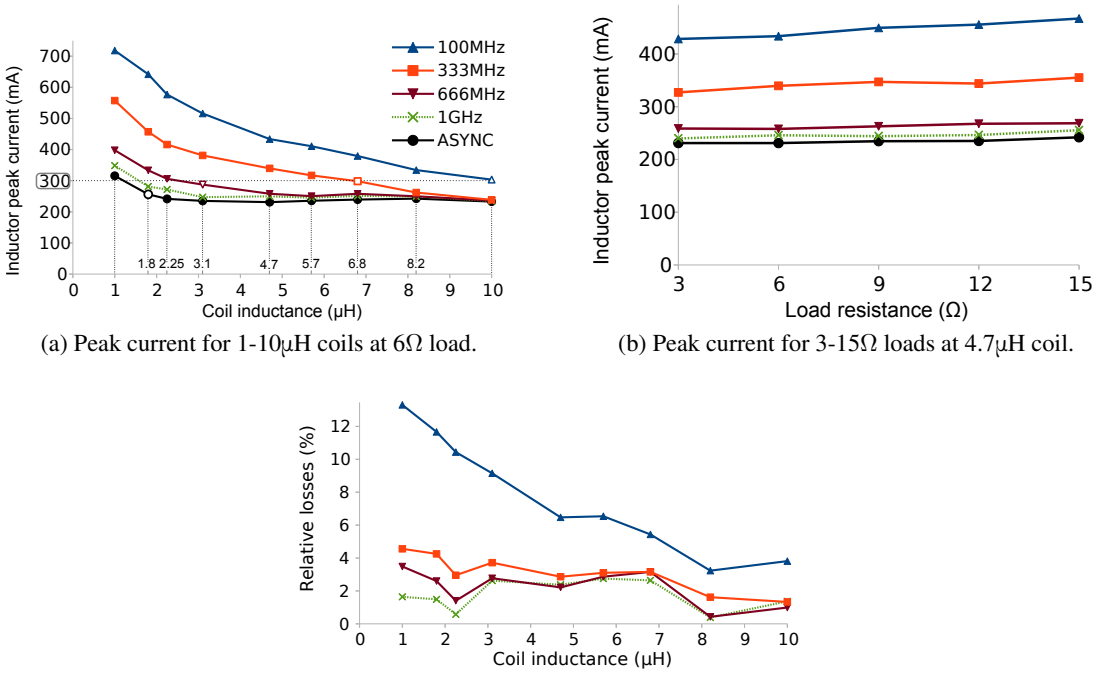


(b) Event-driven asynchronous control.

Figure 2.8: Simulation waveforms.

Controller	HL (ns)	UV (ns)	OV (ns)	OC (ns)	ZC (ns)
100MHz	25.00	25.00	25.00	25.00	25.00
333MHz	7.50	7.50	7.50	7.50	7.50
666MHz	3.75	3.75	3.75	3.75	3.75
1GHz	2.50	2.50	2.50	2.50	2.50
ASYNC	1.87	1.02	1.18	0.75	0.31
Improvement over 333MHz	4x	7x	6x	10x	24x

Table 2.1: Comparison of the reaction time.



(a) Peak current for 1-10µH coils at 6Ω load.

(b) Peak current for 3-15Ω loads at 4.7µH coil.

(c) Losses for 1-10µH coils at 0.55A sink current.

Figure 2.9: Comparison of peak current and inductor losses.

smaller peak current when using the same coils, see Figure 2.9a. This advantage can be efficiently traded off for the size of coils, which are bulky and may affect the dimensions of consumer gadgets. For example, for a 6Ω load, the asynchronous control maintains the peak current below 300mA using 1.8µH inductors, while the synchronous control requires 10µH coils at 100MHz, 6.8µH at 333MHz, or 3.1µH at 666MHz (denoted by hollow markers in Figure 2.9a). This trend persists for a wide range of load resistance that covers the typical computational load of mobile microprocessors, see Figure 2.9b for the peak current data at 3-15Ω loads and 4.7µH coils. Note that smaller peak current translates into fewer losses in the inductors, as shown in Figure 2.9c normalised to the losses of asynchronous buck, and helps to achieve higher power efficiency. These improvements enable a designer to reduce the physical dimensions of the system, while maintaining its operating characteristics.

This work demonstrates clear advantages of asynchronous design methodology for "little digital" control. The simulation results show improved reaction time, voltage ripple, peak current, and inductor losses of the buck when controlled asynchronously. These benefits lead to the higher efficiency of power conversion, and can be traded off for the cost of analog components.

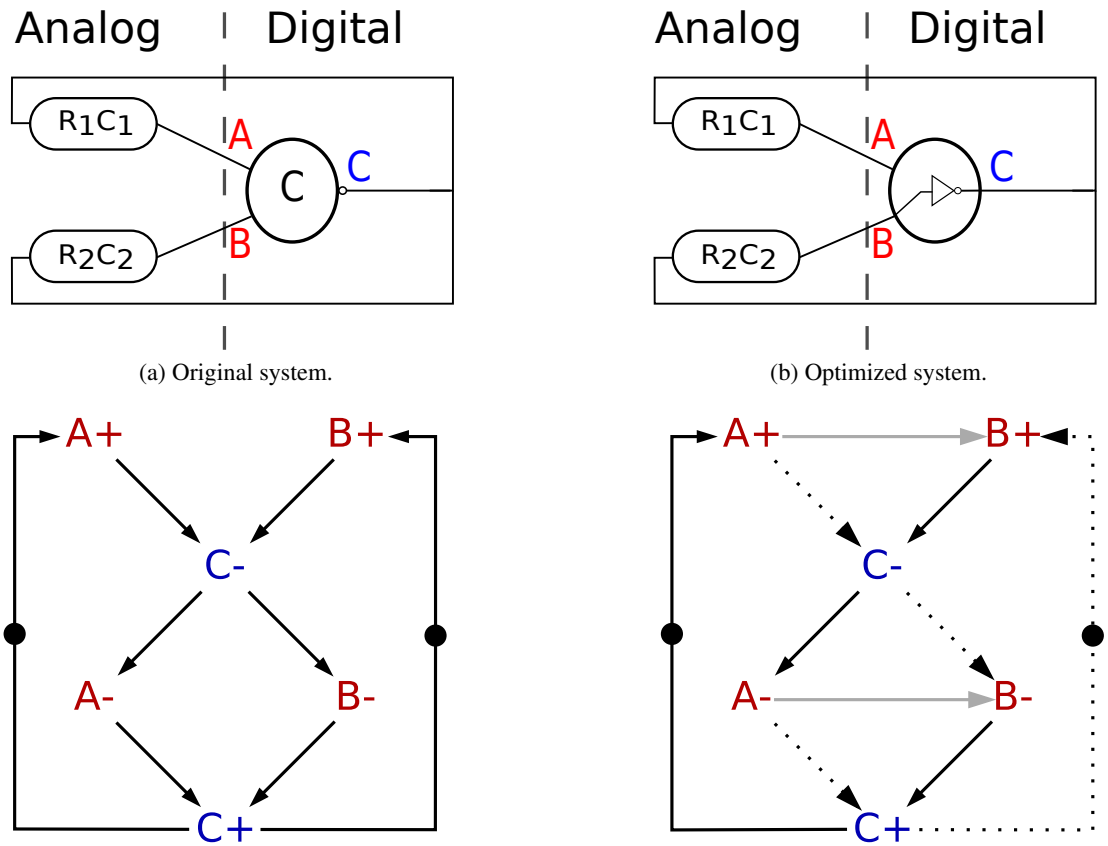
Chapter 3. Formal verification

Asynchronous logic offers a number of improvements for AMS systems. The A4A design flow, introduced in the previous chapter, provides automated means for asynchronous circuit design and synthesis. However, STGs, formalism used in the A4A flow, have no means to describe the behavior of the analog environment, making full-system verification problematic.

Several approaches, such as hybrid automata [37] and hybrid Petri nets [20], have been proposed to construct abstract models of mixed-signal systems [77]. These models enable formal verification methods for AMS designs, reducing the need for the conventional simulation methods and improving robustness of the whole system. One particular example of hybrid Petri nets, *labeled Petri net* (LPN) [44], can specify timing behavior, discrete events, and continuous dynamics. LPNs include continuous variables that can be sampled in an enabling condition and delay assignment labels on the transitions in the LPN. These continuous variables or their rates of change can be modified by transition firings. In addition, conversion between STG and LPN formats can be done in a straightforward manner, thus making analysis of asynchronous control within analog environment with formal methods possible.

As a motivating example, consider the C-element example shown in Figure 3.1a. This AMS system consists of a C-element, which feeds its output through an inverter to two RC circuits with different time constants. Without knowledge of the analog environment, the designer has to use the complete STG specification from Figure 3.1c. However, using the proposed workflow, it is possible to discover that A changes before B, leading to the updated STG with added timing assumptions (shown as gray arcs) in Figure 3.1d. These timing assumptions make other arcs obsolete (shown as dashed arcs) introducing the possibility of control simplification. As a result, it is possible to use an inverter instead of a C-element as shown in Figure 3.1b.

The main goal of this chapter is to introduce the novel automated workflow, which enables formal verification of AMS systems with asynchronous control that has been optimized with correct timing assumptions extracted from the full-system model. The flow combines two state-of-the-art tools, WORKCRAFT and LEMA. WORKCRAFT allows automated synthesis of asynchronous circuits from their STG specification, while LEMA provides means to construct



(a) Original system.

(b) Optimized system.

(c) Original STG.

(d) Optimized STG.

Figure 3.1: C-element example.

abstract models of the entire AMS system via LPN formalism and perform formal verification through reachibility analysis, thus increasing reliability of AMS circuits. The application of the flow is demonstrated on the C-element example and buck converter in Section 3.4.

3.1 Labeled Petri Net

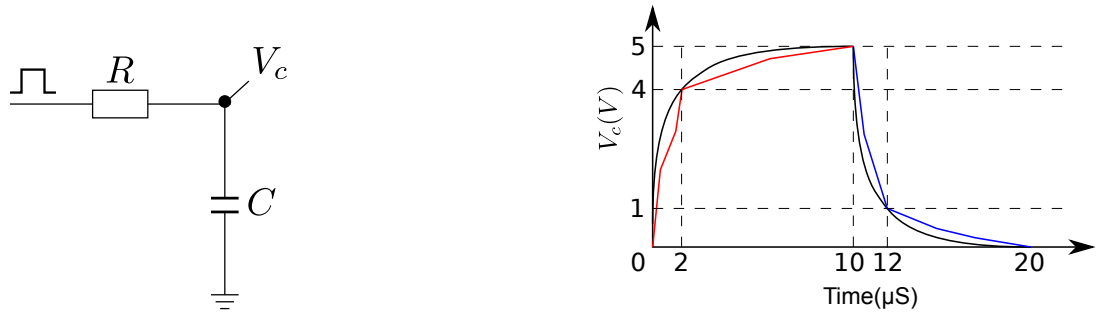
An LPN is a special type of a PN with additional apparatus to capture the dynamics of continuous and analog systems. Similar to a classical PN, an LPN contains places, that keep track of the current state, and transitions, that evolve the LPN from one state to another. In addition LPNs contain Boolean and continuous variables, which model signals in the simulation data. Transitions have extra connotation added to them via labels, which affect the transition firing condition and model variables. In order for a transition to fire, not only do enough tokens have to be present in the preset, but also the enabling condition, expressed as a Boolean formula, of the associated label has to be satisfied. Furthermore, the enabled transition is only allowed to fire in a specified time window. Once the transition fires, it modifies

associated the variables via assignment statements. Formally, an LPN is defined as a tuple [27] $N = \langle P, T, F, M_0, T_F, X, V, Y_0, Q_0, R_0, L \rangle$ where:

- P is a finite set of places;
- T is a finite set of transitions;
- $F \subseteq (P \times T) \bigcup (T \times P)$ is the flow relation;
- $M_0 \subseteq P$ is the set of initially marked places;
- T_f is a finite set of failure transitions;
- X is a finite set of discrete variables;
- V is a finite set of continuous variables;
- $Y_0 : X \rightarrow Z$ is the initial range of values for each discrete variable;
- $Q_0 : V \rightarrow Q$ is the initial range of values for each continuous variable;
- $R_0 : V \rightarrow Q \times Q$ is the initial range of rates of change for each continuous variable;
- L is a tuple of labels defined below.

Places (P) with initial markings (M_0), transitions (T), and arcs (F) connecting them represent traditional PN elements. Failure transitions (T_f) are used by LPNs to signal when a failure has occurred. The labels, L, for an LPN are defined by the tuple $L = \langle En, DA, XA, VA, RA \rangle$:

- $En : T \rightarrow P_\phi$ labels each transition $t \in T$ with an enabling condition;
- $DA : T \rightarrow P_\varkappa$ labels each transition $t \in T$ with an expression for the delay before a transition t can fire;
- $XA : T \times X \rightarrow P_\varkappa$ labels each transition $t \in T$ and discrete variable $x \in X$ with an expression for the discrete variable assignment that is made to x when t fires;
- $VA : T \times V \rightarrow P_\varkappa$ labels each transition $t \in T$ and continuous variable $v \in V$ with an expression for the continuous variable assignment that is made to v when t fires;
- $RA : T \times V \rightarrow P_\varkappa$ labels each transition $t \in T$ and continuous variable $v \in V$ with an expression for the rate assignment that is made to v when t fires,



(a) Schematic. (b) Waveform.

Figure 3.2: RC circuit.

where P_ϕ and $P_\varkappa$ are sets of all formulae that can be constructed using the Boolean grammar, ϕ, and the numerical grammar, $\varkappa$, respectively [10]. The Boolean grammar consists of the Boolean operators: negation, conjunction, and disjunction, the Boolean data types and the relational operator, which operates on the numerical data types. The numerical grammar combines the numerical data types: rational constants, discrete variables, and continuous variables, the mathematical operators: summation, multiplication, and subtraction, as well as special functions *int* and *uniform.* The function *int* converts the Boolean **true** or **false** to an integer 1 or 0, respectively, while the function *uniform*[1] returns a uniform random value between the specified lower and upper bounds.

As an example, consider the RC circuit, that is connected to a square wave generator, as shown in Figure 3.2a. The voltage over the capacitor, being charged and discharged periodically, follows the well-known exponential curve in Figure 3.2b. The corresponding LPN model, Figure 3.3, linearly approximates the waveform via ranges of rates.

The LPN model consists of model's variables, initial conditions, which set values of variables, and a Petri net model with additional label semantics, added to the transitions. In the provided model the output voltage(V_c) and its rate($V_c^{'}$) are set to 0. The operation of the model is similar to the operation of a classic Petri net model and for a transition to fire tokens have to be present in all preset places of a transition. Furthermore, a transition can only fire if it's enabling condition, which is expressed as a Boolean formula, is evaluated as **true**. For example, The charging of the capacitor is initiated by the transition *Charge 1.* The enabling condition of the transition in curly brackets is set to *true*, which effectively means that the transition does not perform any checks. In contrast the enabling condition of the transition *Charge 2* can only fire if the voltage V_c reaches the value of 4000 mV. Note that enabling conditions of transitions

[1]To improve the readability of LPN models the *uniform* statement is replaced by square brackets. E.x. *uniform(1,2)* is equivalent to [1,2].

Discharge 2 and *Return* have logical negation which effectively flips the comparison sign from $\geq$ to $\leq$.

However, the transition cannot fire immediately as it is only allowed to fire in the window, specified by the delay expression in square brackets. For example, the transition *Charge 1* has a delay between 1 and 2 microseconds. Once the transition fires it modifies variables as stated in the assignment expression. In the case of the transition *Charge 1* a positive rate to the output voltage in the range between 1000 and 2000 $mV/\mu S$[2] is assigned. This range of rates effectively represents a whole spectrum of possible rates within allowed boundaries, which allows it to capture multiple rate trajectories, using only one statement. One such possible trajectory is shown in Figure 3.2b via red and blue lines, that approximate the original curve, during charge and discharge cycles, respectively.

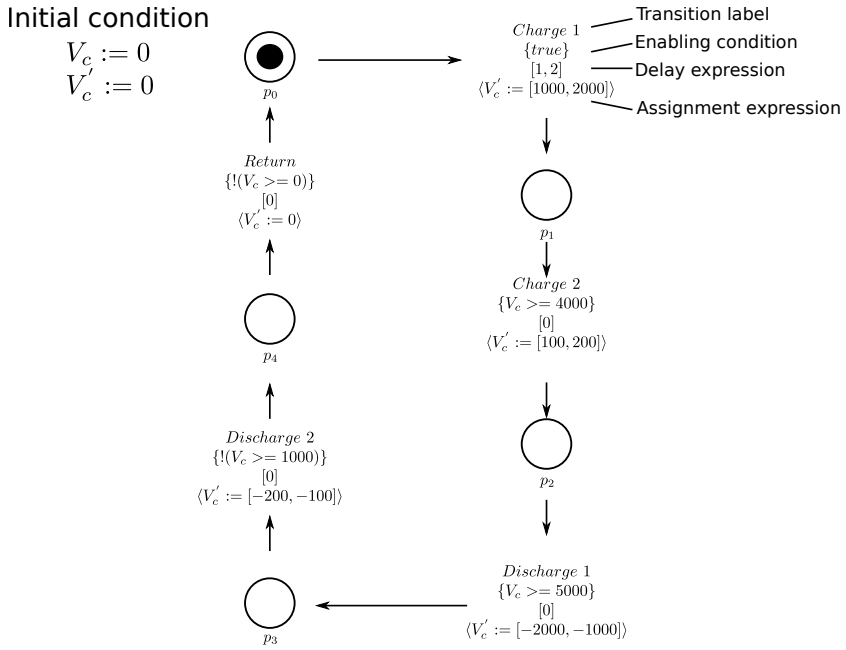


Figure 3.3: RC LPN model.

3.2 LEMA

LEMA [28], the *LPN Embedded Mixed-Signal Analyzer*, is a tool that seeks to enable the formal verification of AMS circuits and has a tool flow as shown in Figure 3.4. The flow begins with the creation of an LPN model from a simulation trace. The model is combined with a set of properties to be used in the formal verification process. More information about the flow aspects is presented in the subsequent sections.

[2]Values of real signals have been upscaled by a factor of 1000 due to integer value requirements.

The tool has been utilized to verify a number of AMS circuits including digital to analog converters (DACs), phase interpolators, voltage controlled oscillators [39], etc. In [73] the tool has been successfully used to model the behavior and verify the design of a switched capacitor circuit. Formal verification underlined the problem with charge building up on a capacitor, while stochastic simulation did not present such a result. Furthermore, LEMA has shown potential in analyzing non-linear systems, such as a tunnel diode oscillator [44].

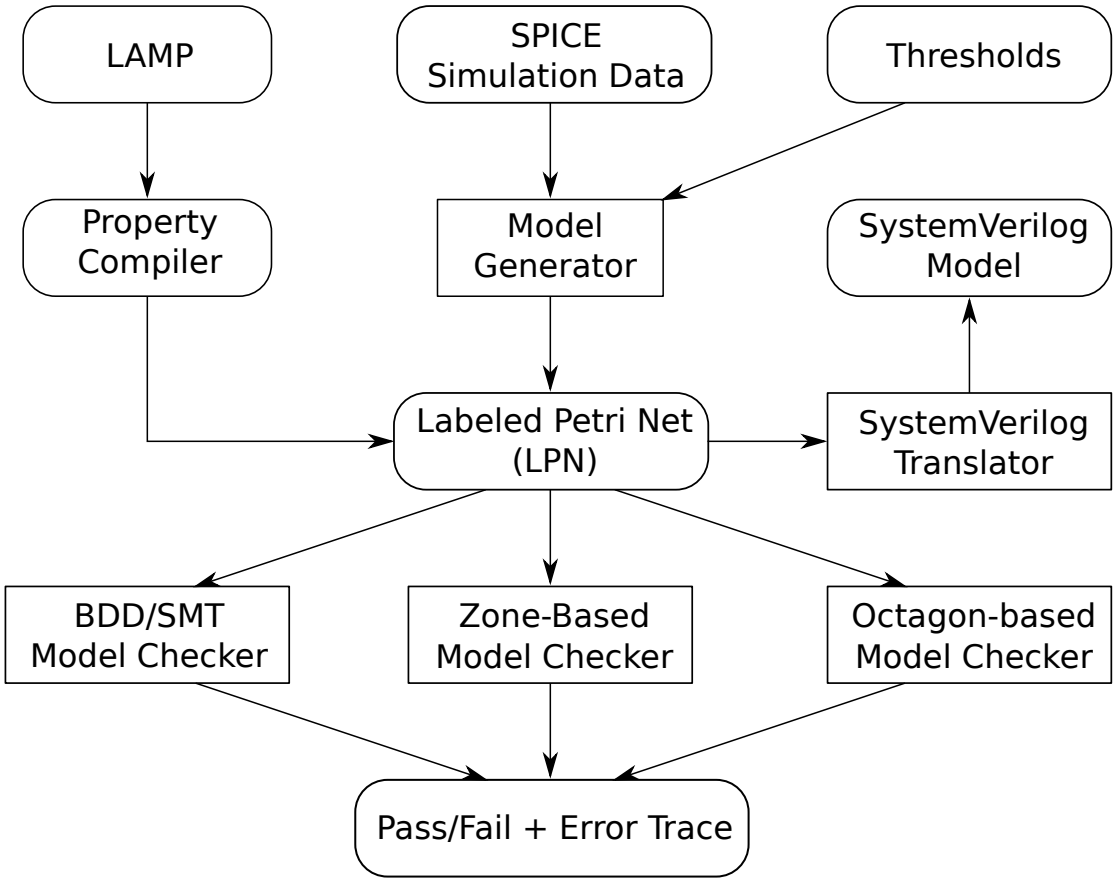


Figure 3.4: LEMA's tool flow.

3.2.1 *Model generator*

In order to use formal verification, it is necessary to have methods for automated construction of formal models. While it is possible to create LPNs by hand, it is a tedious process, and it is not easy to convince AMS designers to do so. Consequently, LEMA streamlines the process of constructing LPNs by providing a means for automated generation of models from simulation traces. The process starts with the analysis of the existing SPICE simulation data [45, 10, 38], used in a traditional simulation-based verification approach. The data is discretized, using a set of thresholds, provided by the user or automatically generated by the tool. The thresholds

divide the space of the continuous variables into regions, which are used by the model generator operates to produce a formal model in LPN format [44]. Furthermore, the model generator can identify discrete transitions and assign an appropriate delay. More information on the model generation aspects is presented in Chapter 4

3.2.2 *Property expression*

After a formal model has been created, the next step is to create a property to check the desired behavior. The properties are specified using the *Language for Analog/Mixed-Signal Properties* (LAMP) [27]. This language provides convenient means for a designer to express verification properties without the knowledge of LPN semantics. The specified property is compiled into an LPN to be later used by a model checker.

Algorithm 3.1 A before B: LAMP property.

```
property AbeforeB {
  real A;
  real B;
  always{
    assertUntil(~(B>=2500),(A>=2500));
    wait((B>=2500));
    assertUntil((B>=2500),~(A>=2500));
    wait(~(B>=2500));
  }
}
```

For example, to optimize the C-element example, introduced earlier, it is necessary to determine the order of signals A and B. To do that a verification property, shown in Algorithm 3.1, has been created. This property checks that signal A changes before signal B. Initially, both signals are set to 0 and an assertion is made that ensures that signal A goes high, while signal B stays low[3]. Once A is high, the property waits for signal B to go high. After that, a similar check is performed to confirm that A goes low before B. The compiled LPN for this property is shown in Figure 3.5.

3.2.3 *SystemVerilog translator*

A property can be checked using a system-level simulation. For that reason, LEMA can encode LPN models into the *SystemVerilog* (SV) format. In SV, places become *logic* variables and

[3] Although, A and B are discrete signals, the property operates on continuous variables and explicitly discretizes analog signals with 0V being low and 5V being high.

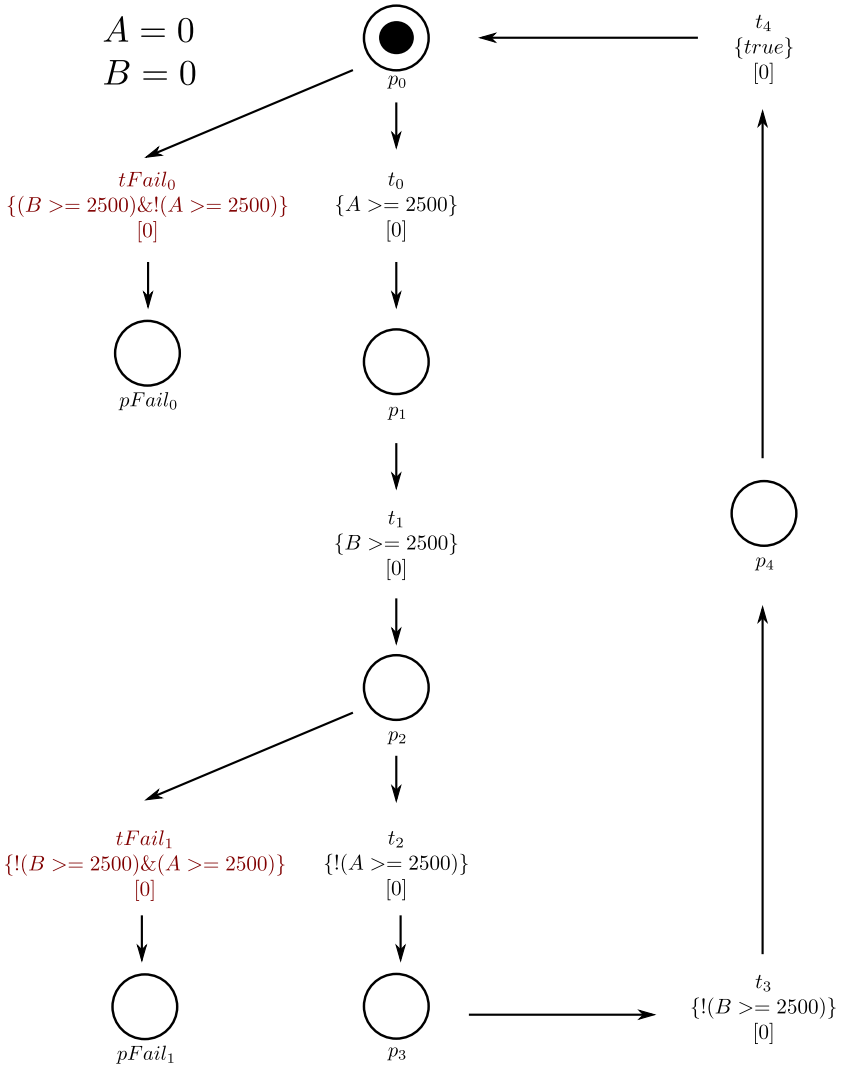


Figure 3.5: A before B: LPN property.

transitions become wires, while LPN variables are translated into *real* variables. The value of the logic variables equates to the place being not marked for logical zero or marked for logical one. The initial state of the system is set by the SV *initial* block, which assigns values to logic variables and real variables in accordance to LPN initial marking and initial conditions. A transition firing consists of two steps. Once the enabling condition is satisfied, the corresponding wire is set high after a specified delay. After that, a SV *always* block, triggered by the positive edge on the wire, executes the assignment expression of the LPN transition and sets the new net marking by changing the real and logic variables. A fragment of SV code for an RC circuit model is shown in Figure 3.2. Note, that *delay* and *uniform* are custom SV functions that have the same functionality as LPN statements.

Algorithm 3.2 Portion of the SystemVerilog for the RC model.

```
module rc_model ();
        wire charge1, charge2, discharge1, discharge2, return;
        real Vc_rate,Vc;
        logic p0, p1, p2, p3, p4, fastClk,reset;

        initial begin
                Vc_rate = 0; Vc = 0; reset = 1;
                p0 = 0; p1 = 0; p2 = 0; p3 = 0; p4 = 0;
                #1;
                p0 = 1; //Initially Marked
                reset = 0;
        end

        always #1 fastClk = (~fastClk)&(~reset);
        always @(fastClk) begin
            Vc <=Vc+Vc_rate;
        end

        assign #(delay(~charge1,1,2)) charge1 = p0;
        assign #(delay(~charge2,0,0)) charge2 = p1 && (Vc>=4);
        ...
        always @(posedge charge1) begin
                p0 <= 0; p1 <= 1;
                Vc_rate <= uniform(1.0,2.0);
        end
        always @(posedge charge2) begin
                p1 <= 0; p2 <= 1;
                Vc_rate <= uniform(0.1,0.2);
        end
        ...
endmodule
```

3.2.4 *Model checker*

LEMA provides multiple reachability analysis methods. It includes both an explicit state method, originally used for timing verification that leverages *zones* and *difference bound matrices* (DBMs) to represent the continuous state space [44], and implicit state methods, that use *binary decision diagrams* (BDDs) and SMT solvers [73]. The main verification method, used in this work, is based on the zone-based model checker.

Zones are a subset of convex polytopes with 90° and 45° angles between half-planes. Thanks to this limitation a zone can be completely determined by the pairwise inequalities $v_i - v_j \leq c_{i,j}$ together with $v_i - t_0 \leq c_{i,0}$ and $t_0 - v_i \leq c_{0,i}$, where $v_0, ..., v_{n-1}$ is a set of continous variables, $c_{i,j}$ is a constant and t_0 is a timer that is always zero. A compact representation of

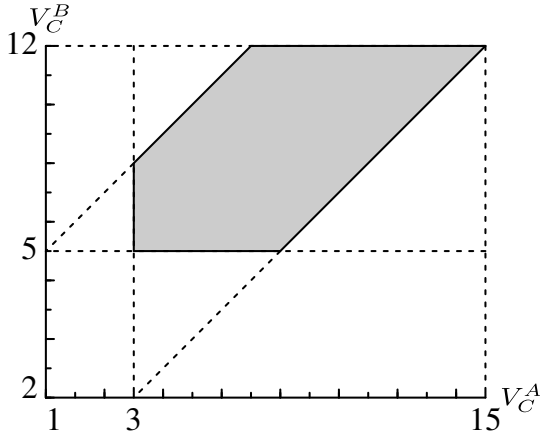


Figure 3.6: C-element zone.

these inequalities is created by collecting constants $c_{i,j}$ into a DBM. A standard zone, obtained from the reachibility analysis of the C-element example, is depicted in Figure 3.6. This example has two continuous variables V_C^A and V_C^B and the corresponding inequalities are:

$$\begin{array}{lll} t_0 - t_0 \leq 0 & t_0 - V_C^A \leq -3 & t_0 - V_C^B \leq -5 \\ V_C^A - t_0 \leq 15 & V_C^A - V_C^A \leq 0 & V_C^B - V_C^A \leq 3 \\ V_C^B - t_0 \leq 12 & V_C^A - V_C^B \leq 5 & V_C^B - V_C^B \leq 0 \end{array}$$

Collecting constants gives corresponding DBM:

$$\begin{array}{c} \begin{array}{ccc} t_0 & V_C^A & V_C^B \end{array} \\ \begin{bmatrix} 0 & -3 & -5 \\ 15 & 0 & 3 \\ 12 & 5 & 0 \end{bmatrix} \end{array}$$

Initially, zone-based methods were used to verify timed automata, which operated only clock variables. To cope with continous variables, which can have rates not equal to one, LEMA uses warping [44]. With warping, variables are scaled by their rate, turning them into rate-one variables and then the resulting figure is over-approximated by a zone.

3.3 Combined verification environment

Asynchronous and formal verification tools, described in the previous chapters, seek to improve the design of AMS systems. However, there are no convenient means to exchange data between them. As a result, the verification process becomes cumbersome and does not provide any information on optimization possibilities for the asynchronous digital control circuit.

LEMA and WORKCRAFT were identified as the most suitable candidates for the creation of a unified verification and synthesis environment. Both tools are based on PNs formalism,

so the data transfer process is straightforward. LEMA provides excellent capabilities for model generation from simulation traces, which appeals to analog designers, who are not familiar with PNs. It is also important to note, that these tools are actively developed and are written in the same programming language, which simplifies the integration process. A workflow, combining both tools, is presented on Figure 3.7.

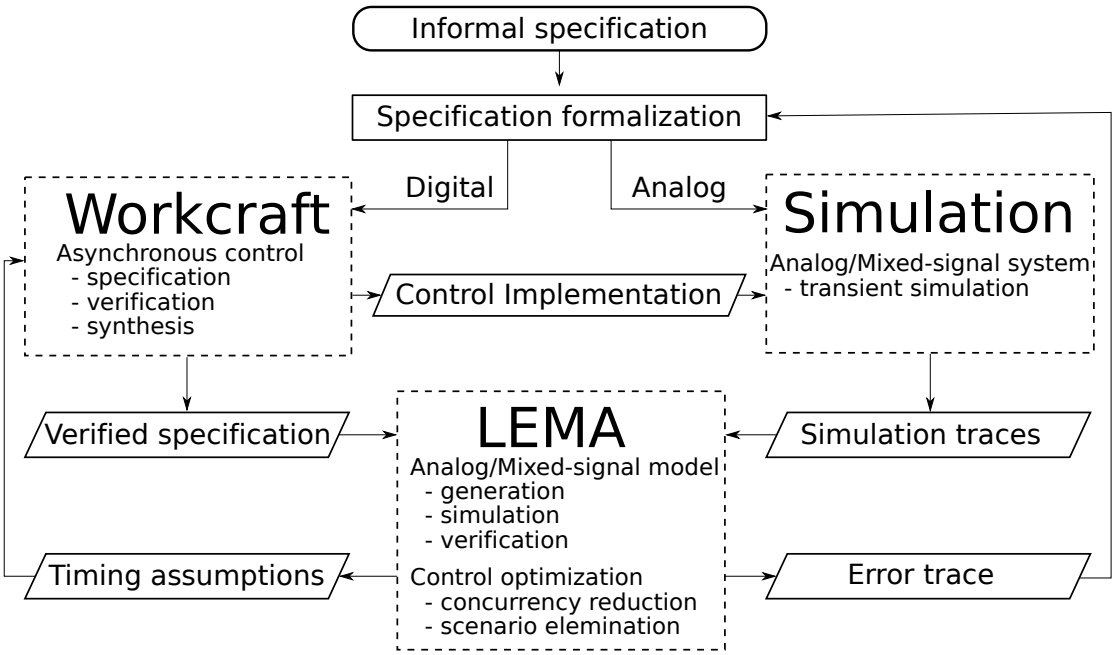


Figure 3.7: Workcraft and LEMA joint workflow.

Initially, only an informal specification of the AMS system is given. This specification contains high-level information on system behavior and does not represent internal structure. The first step is system's formalization: digital part is expressed in STG format, using WORKCRAFT framework, and analog environment is implemented as a behavioral model, for example in the form of Verilog-AMS or Spice.

The digital module is developed in accordance to the A4A design flow, described in Chapter 2. The control STG is verified, using one of the back-end tools, and the later control circuit is synthesized from the verified specification. The resulting verilog netlist is combined with the model of the analog environment.

In order to proceed with formal verification of the whole systems, it is necessary to obtain a number of simulation traces. There are two possible ways to do so:

- **Full-system simulation.** Verilog netlist, obtained from the STG, is combined with the environment model and the system undergoes transient simulation. Although this is a an easy and straightforward method, it has a serious drawback: the resulting trace might not

cover all possible states of the environment, making verification limited.

- **Simulation of individual modules.** The analog part of the design is split into individual modules, which are simulated intensively to provide better state coverage. However, special care has to be taken during system partitioning to prevent space state explosion, while keeping sufficient level of detail.

After the simulation data is obtained, LEMA is able to generate LPN models for the analog portion of the AMS system. The STG is automatically converted to the LPN format. As a result, hybrid Petri net model of the entire system is created, thus enabling the checking of important properties of the design. If formal verification reveals no problems, then digital control can be optimized by comparing states in a stand-alone STG and the one used in combination with the analog environment. Unused states can be removed, reducing the complexity of the control circuit. Discovered optimizations are reported to WORKCRAFT in the form of timing relations for the input signals and the digital part can be resynthesized, using the information about correct timing assumptions. When no optimizations are possible, the designer can proceed to layout implementation.

The proposed methodology aims to integrate both tools in a unified environment and to provide a joint workflow for the synthesis and verification of AMS systems with asynchronous control.

3.4 Buck control

The described methodology has been applied to the asynchronous control module of a buck converter. The circuit, synthesized from the STG specification, is used in mixed-signal simulation with an analog environment. Using the set of generated simulation traces, a full-system model was generated, which shows possibilities for concurrency and scenario optimizations in the original specification.

3.4.1 *Buck converter*

DC-DC converters are an important part of modern digital circuits and are required to provide a stable power supply over long periods of time. A basic power regulator consists of an analog block and a digital controller, as shown in the schematic in Figure 3.8. The controller determines the state of NMOS and PMOS transistors as a reaction to *acknowledgement* (ACK), *under-voltage* (UV), and *over-current* (OC) conditions. These conditions are detected

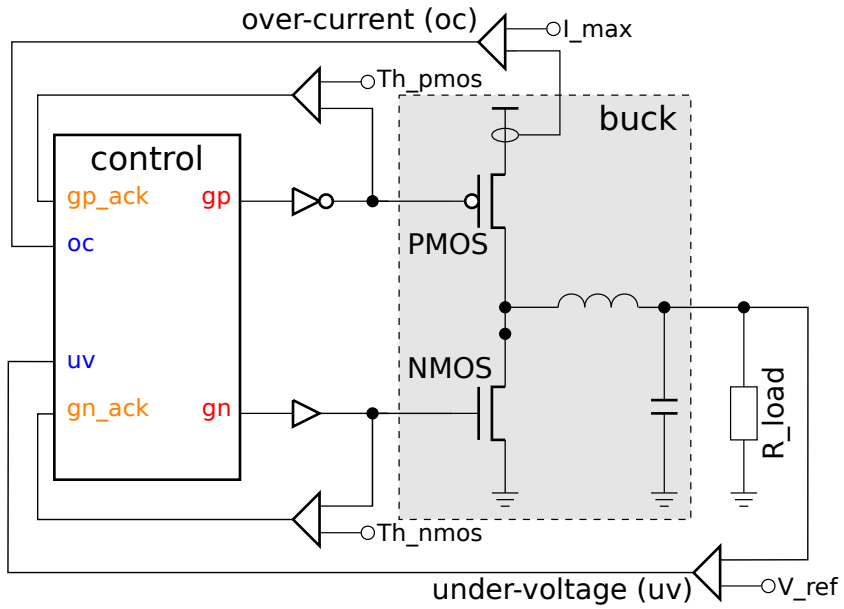
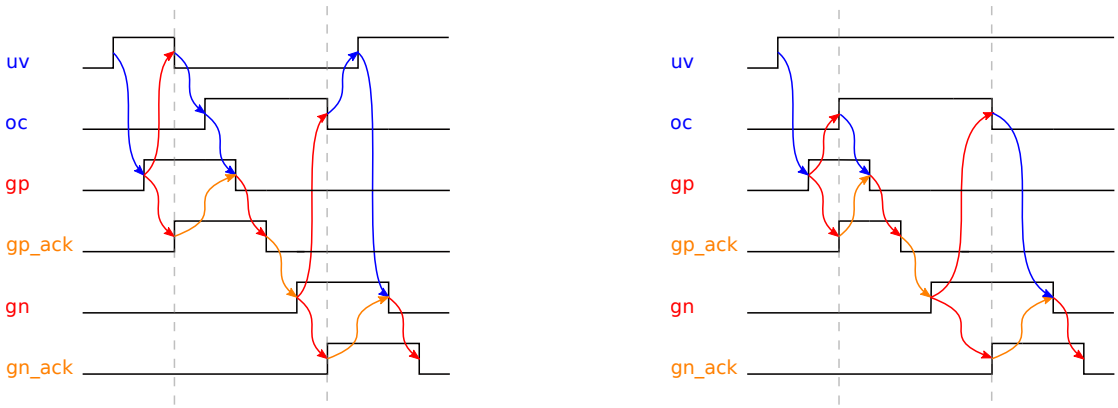


Figure 3.8: Buck converter schematic.



(a) OC with UV.

(b) OC without UV.

Figure 3.9: Informal specification.

and signaled by a number of special sensors, implemented as comparators in combination with buffering latches.

Initially, specification of the control module is given as timing diagrams with causal relations between signals. Two possible scenarios are considered: stable state when output capacitor charges up to threshold value during one charge cycle (see Figure 3.9a), and start-up operation mode during which multiple charge cycles are needed to charge the output capacitor (see Figure 3.9b).The formal specification is derived from the provided diagrams. The resulting STG, shown in Figure 3.10, captures the behavior of both scenarios. In addition, special care is taken to incorporate the concurrent nature of the transistors' acknowledgments and over-current signals.

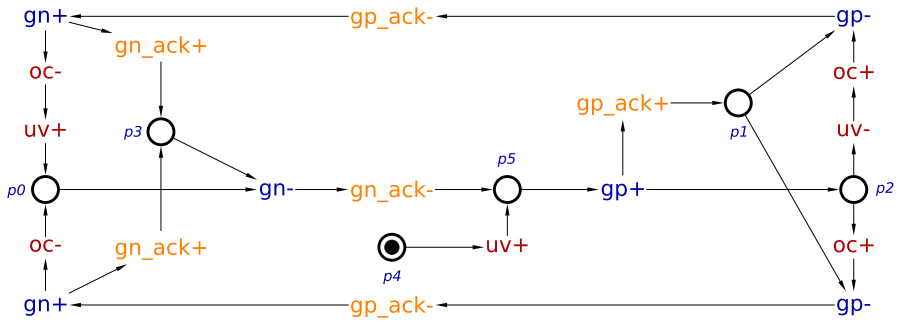


Figure 3.10: Buck control STG.

3.4.2 *Model generation*

The control circuit, synthesized from the STG, is combined with a Verilog-AMS model of the analog part of buck converter to undergo a series of simulations, using the VIRTUOSO AMS simulation environment. The dynamic resistive load is used to ensure that the system works under different operating conditions.

In order to generate abstract models of the analog components, causal relations between the digital and the corresponding analog signals have to be established. There are two possible types of causality that have to be identified:

- **Direct causality**: An analog signal is directly affected by a digital control signal.
- **Indirect causality**: A digital control signal affects an analog signal transitively via some intermediate events.

The voltage on the transistors' gates is in direct correlation with control outputs *gp* and *gn* and determines the state of the corresponding acknowledgment signals *gp_ack* and *gn_ack*, as shown on Figure 3.11. The process of model generation revolves around determining states with unique variable encodings. Values of continuous variables are assigned to different regions or *bins*, according to the specified thresholds and linearly approximated with ranges of rates. The construction of the LPN is performed by creating transitions with proper variable assignments and guard conditions for input signals. These transitions are linked together in accordance with their evolution in the waveform.

The resulting LPN model shown in Figure 3.12 captures the presented behavior for signals of the PMOS transistor. The model decides upon voltage evolution rate, depending on input signal state and current voltage value. Transitions *charging{1,2}* and *discharging{1,2}* represent charging and discharging processes of the gate capacitor with threshold points specified at the change of the acknowledgment signal. For example, the annotation of the *discharging1* transi-

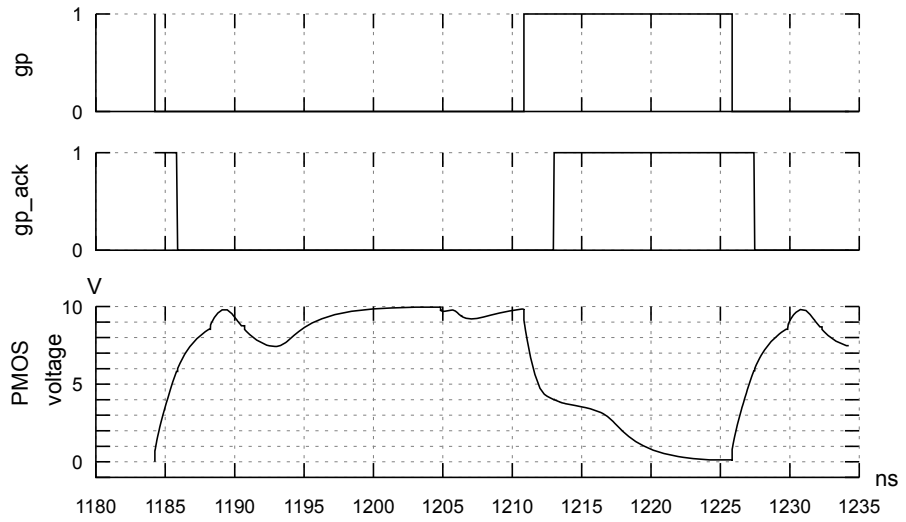


Figure 3.11: PMOS acknowledgement signals.

tion means that when the *gp* is true and the voltage of the *gp_gate* signal is above or equal to 4V, the change of this voltage must be relatively fast (within the shown range between -0.27V/ns and -0.28V/ns), and then the *discharging2* shows that as soon as the *gp_gate* voltage drops below 4V the rate of discharge becomes slower (within the shown range between -0.03V/ns and -0.048V/ns). Special transitions *corner{1,2}* are essential to prevent voltage from reaching values not present in the original waveform. The model of the acknowledgment signal (upper part of the Figure 3.12) communicates with the voltage model (lower part of the Figure 3.12) via guard conditions and assigns output values in accordance with the voltage value. For example, the *discharge 2* transition is synchronized with the *ack_pos* transition and *charge2* with *ack_neg* (cf. corresponding events in the waveforms of Figure 3.11). This unique feature of LPNs allows one to construct complex models as a set of small Petri nets with implicit communication via guard variables. A model of the acknowledgment signal of the NMOS transistor is derived in a similar manner.

There is, however, no direct dependency (see Figure 3.13) between digital outputs and the signals responsible for generating UV and OC inputs. The output voltage, as well as current through PMOS transistor, are affected by the inductor current. Thus, an intermediate model of the inductor has to be created first. The model presented in Figure 3.14a describes the behavior of the inductor's ripple current. Transitions *increasing* and *decreasing* set the current rate according to the transistor's state. Over-current and under-voltage models are derived in a similar style to the acknowledgment models with inductor current as one of the input signals. The UV model (see Figure 3.14b) sets output capacitor charge rate, depending on inductor current. A wide range of rates is needed to model dynamic load.

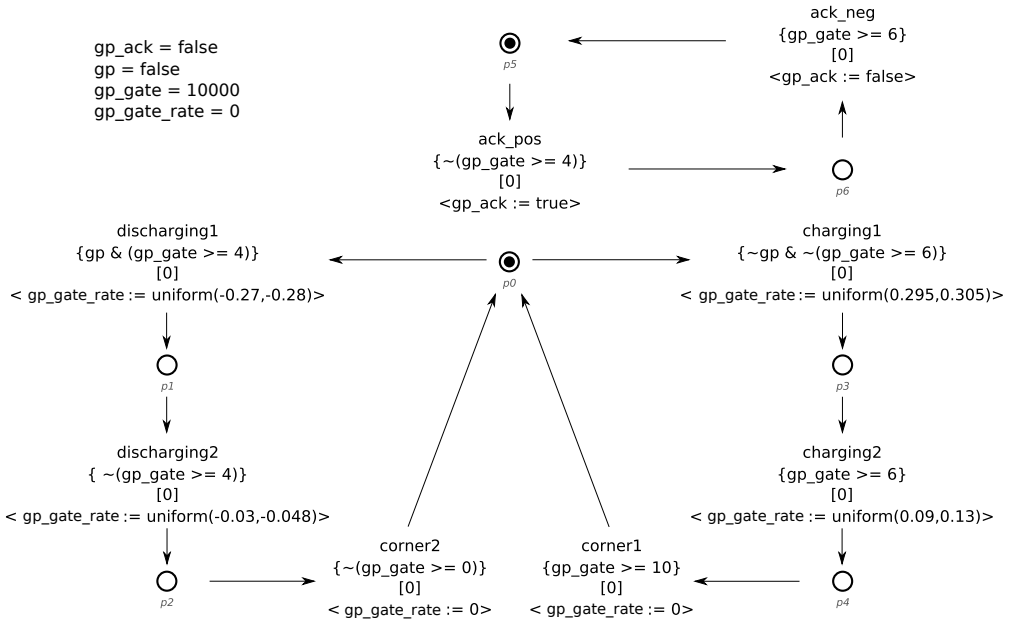


Figure 3.12: PMOS acknowledgement model.

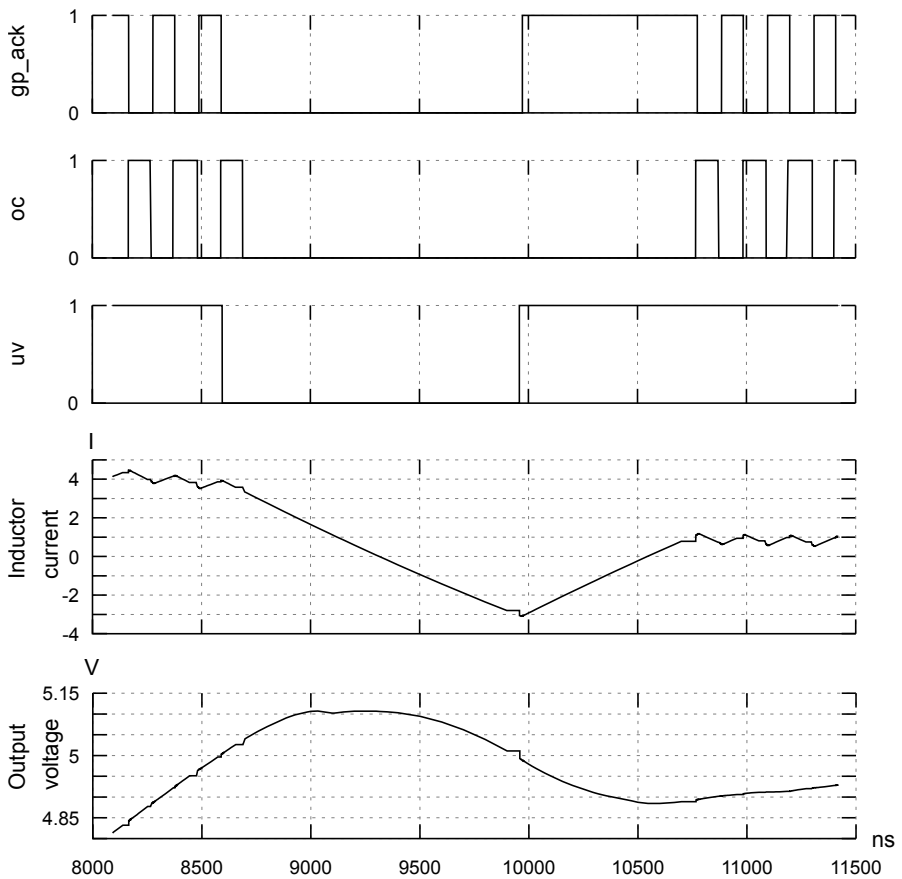
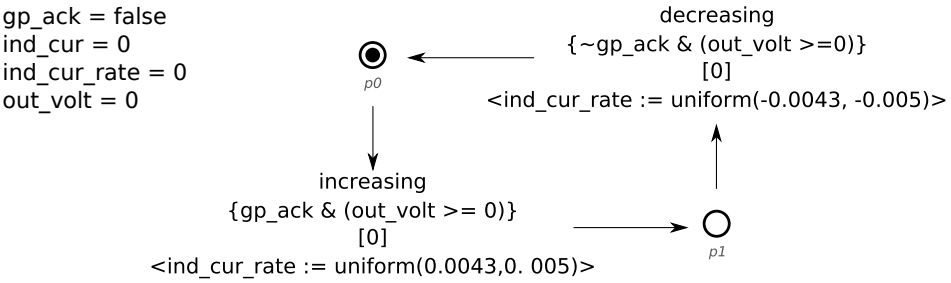
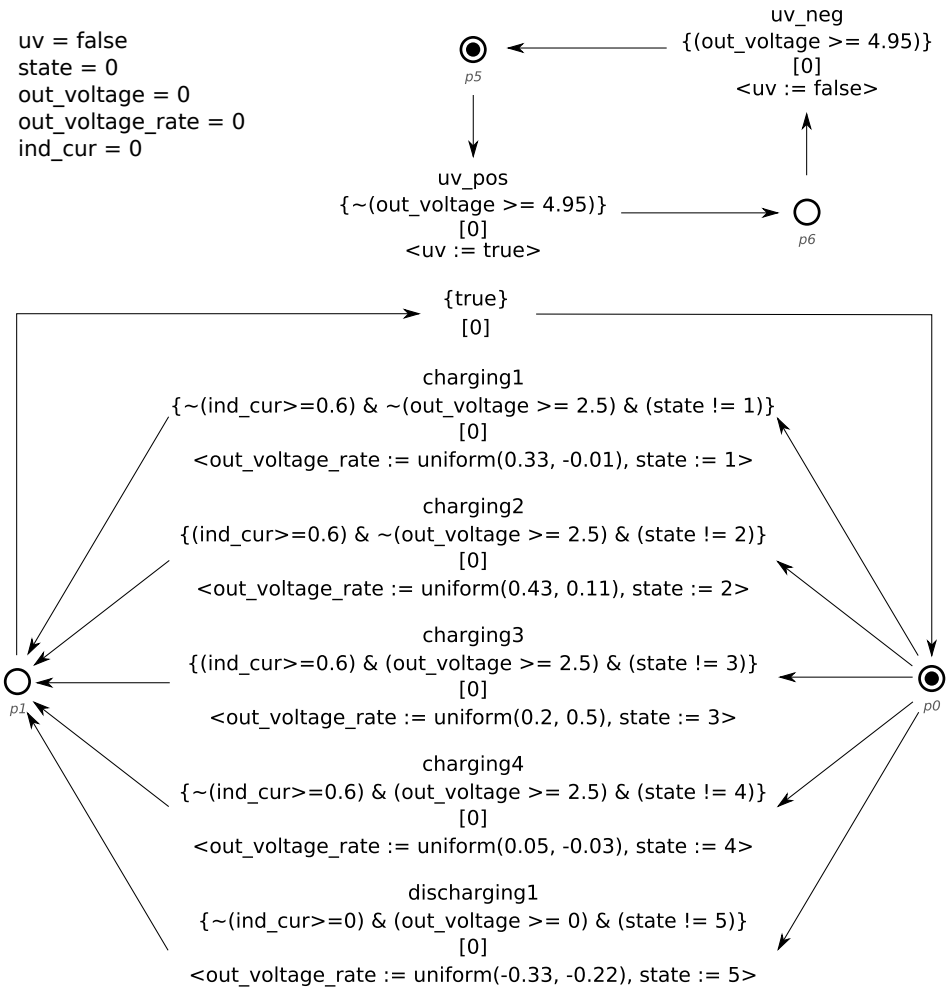


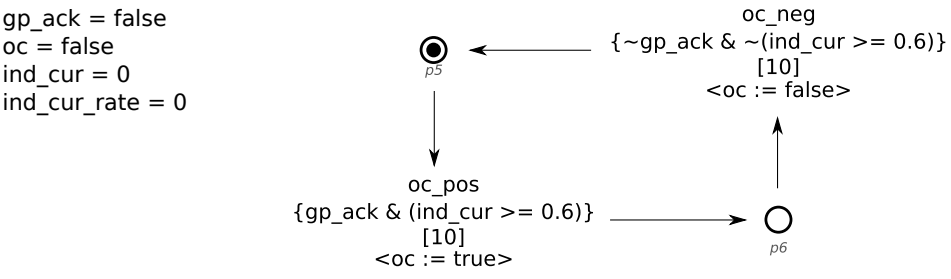
Figure 3.13: Over-current and undervoltage signals.



(a) Inductor model.



(b) Undervoltage model.



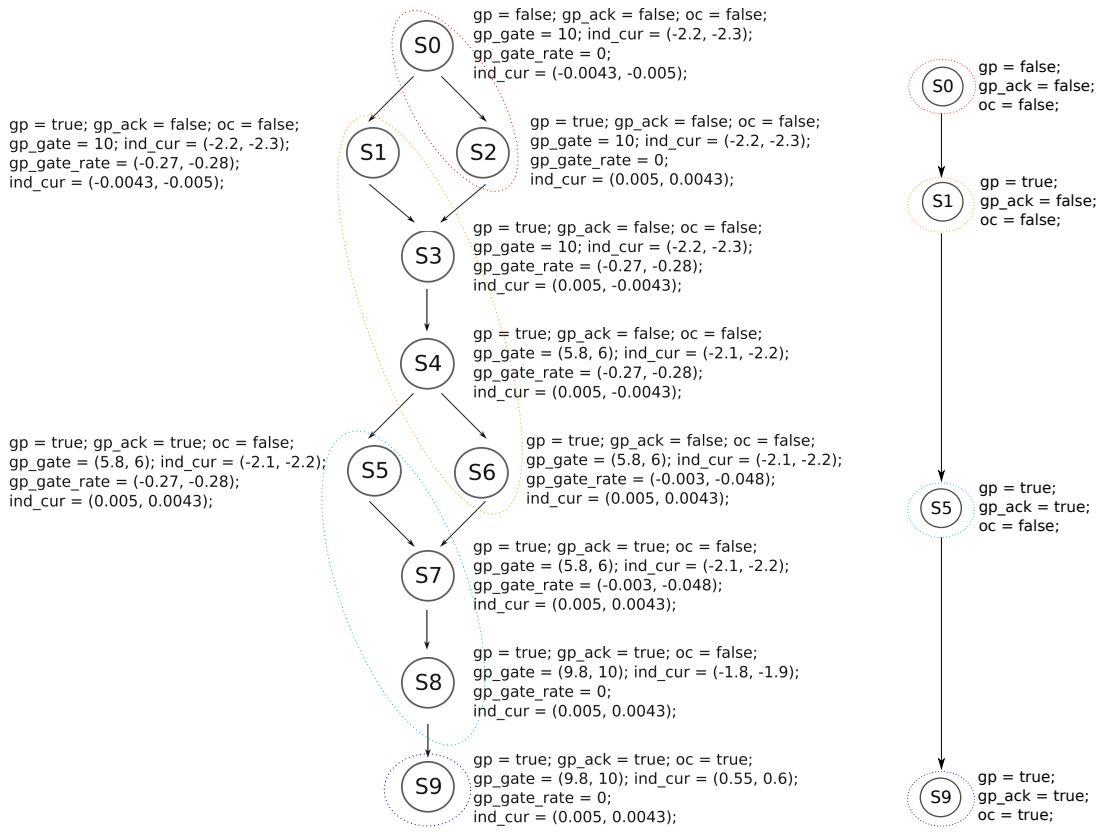
(c) Over-current model.

Figure 3.14: Over-current and undervoltage models.

3.4.3 *Optimization method*

Once the models of the analog blocks are created, it is possible to obtain a full-system model by directly converting the control STG into LPN format. The resulting model can be used to find possibilities for optimization in the control module.

As a first step, the state graph of the system is reduced via a node contraction algorithm. Connected states with similar vectors of control variables are merged together to reduce the state graph size while maintaining the original graph structure. An example of this reduction process is shown in Figure 3.15. Initially a full-state graph is traversed and states, where digital control signals (*gp*, *gp_ack*, *oc*) do not change, as indicated by the dotted ovals, in comparison with all of the preset states, are marked. After that, these marked places are removed, and extra arcs are added to the remaining places to preserve the original graph structure.

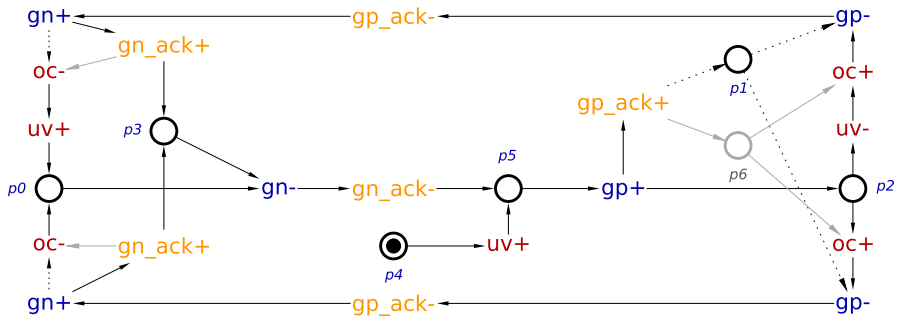


(a) Full state graph.

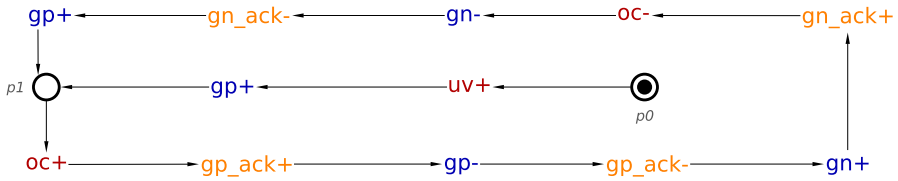
(b) Reduced state graph.

Figure 3.15: State graphs

The reduced state graph is later analyzed to determine the timing relations of the input signals. This timing information can be used in the synthesis process with PETRIFY. Additionally, the reduced state graph can be converted into a STG form, which can be used in



(a) Concurrency reduction.



(b) Scenario elimination.

Figure 3.16: Optimized control models.

Buck control	Total area	Average delay
Original	240	0.83
Reduced concurrency	144	0.58
Removed scenario	112	0.57

Table 3.1: Optimization results.

WORKCRAFT using the established design flow. Although the original structure of the STG may not be preserved, the new version can show greater optimization potential in the form of scenario elimination.

3.4.4 *Results*

The proposed workflow has been applied to the optimization of the buck converter control. The optimization method yields a timing dependency between transistors' acknowledgment and over-current signals, thus reducing all concurrent places in the original model (see Figure 3.16a). While the achieved area and latency reduction are not considerable, these results are achieved in an automated manner, thus promising greater results for more complex projects.

Additionally, a special environment with a small buck capacitor is used to identify extra optimization capabilities. The small capacitance ensured that only one charge cycle is needed for the output voltage to reach the threshold value. As a result it is possible to eliminate scenario in the original STG (see Figure 3.16b).

Optimized models are synthesized using PETRIFY and compared against the original model. Results, reported in abstract units, are summarized in the Table 3.1.

Chapter 4. Model generation

Automated model generation is a relatively new research field [33] and has multiple areas of application, such as simulation models for physical systems, biochemical processes, and manufacturing. The abstract models produced by LEMA's model generator are intended to be used in system-level simulations to verify properties such as connectivity between the digital and analog circuits [24] or for use in formal verification [44]. These models are therefore designed to abstract unnecessary details in order to make the model generation and simulation computationally feasible.

The construction of a buck converter model, described in the previous chapter, revealed a number of limitations in the existing model generation module within LEMA. Due to these limitations, the automatically generated models did not reflect all necessary features of the original simulation traces and had to be manually improved, which rendered the application of the flow less appealing. Specifically, the existing algorithm produces overfit models, which do not demonstrate correct behavior, when used with a different control module. Furthermore, the algorithm operates on the explicitly specified discretization thresholds, which requires extra foresight from the designer and undermines the automated aspect of the design flow. In this chapter, the underlying issues of the existing model generation approach are analyzed, and an improved model generation framework is presented. As evidenced in Chapter 5, the proposed method is capable of producing more general models than the existing method, while requiring less user input.

4.1 ModelGen: existing approach

LEMA's models generator does not attempt to maintain transistor-level accuracy of the original circuit but rather model ranges of parameters and conditions using non-determinism [43]. The resulting abstract models are to be used for system-level verification to check interaction between the digital and analog circuits. For example, one of the possible verification goals is to test the negative feedback loop of the system. Specifically, the model generator trades precision for system scale and model size.

The model generator constructs an LPN model from a set of simulation traces. However, the quality of the model is directly related to the simulations used to create it. Consequently, depending on the dataset provided, the model may not exhibit the full behavior of the system. In this case, there is a possibility that a failing behavior, present in the original system, is not included in the generated model. To overcome this problem the generator can interpolate data and produce general models via a functional approach [10].

A high-level representation of the current model generator is given in Algorithm 4.1. The algorithm operates on a existing LPN N and a set of system variables V. Every variable $v \in V$ is a pair $\langle c, \Sigma \rangle$, where c specifies variable type and $\Sigma : (\sigma_i)_{i=1}^{N}$ is a simulation trace, defined as a finite sequence of data points, where every data point, σ_i, is a pair $\langle \tau, \upsilon \rangle$ with τ being the timestamp[1] and υ being the point value. In addition, the algorithm requires a set of parameters *par*, subsets of which are used in the internal functions.

The model generation operation can be summarized into two stages: discretization of simulation data or data *binning*, and model generation based on the obtained bins. The algorithm begins with the detection of discrete-like signals to report a set of variables $X \subseteq V$, which can be used to approximate the original signals by discrete transitions. The next step is the generation of thresholds Θ for the given data set, based on the user provided parameters. These thresholds are used to assign data points into bins B, which associate every data point with a state vector.

Once the data has been discretized the algorithm proceeds to further process variables and update the LPN. For every variable, depending on the variable type, values Δ, that approximate the generated bins, and bins durations, T, are calculated. This information is used to expand the LPN with additional places and transitions. After all the variables have been processed the LPN is updated with auxiliary transitions, called pseudo-transitions.

The following sections provide a description and analysis of the algorithm functions.

4.1.1 *Data binning*

The purpose of the data binning is to assign each data point of a simulation trace to a bin, which can be approximated with either a discrete value or a linear function, using ranges of rates. To achieve this, the discretization algorithm automatically detects digital-like signals and calculates rates for continuous ones.

Before the simulation data can be used to create an LPN it needs to be compactly represented as a finite set of discrete states to reduce the analysis complexity. The current discretization method relies on grouping data points using a set of thresholds [45]. These thresholds can

[1] It is implied that time ordering of the data points follows element ordering in the region.

Algorithm 4.1 Model generation algorithm.

```
1:  function GENERATEMODEL(N, V, par)
2:      Input:
3:          N: existing LPN or nothing
4:          V: set of system variables
5:          par: a set of parameters
6:      Output:
7:          N: updated LPN or new LPN
8:
9:      Θ := GenerateThresholds(V, par)
10:     B := AssignBins(V, Θ)
11:     X := DetectDMV(V, par)
12:     Δ := ∅, T := ∅
13:     //Calculate variables values in bins
14:     for v ∈ V do
15:         if v ∈ X then
16:             Δ := Δ∪ CalculateValues(v, B, Θ, par)
17:         else
18:             Δ := Δ∪ CalculateRates(v, B, Θ, par)
19:         end if
20:         T := {T}∪ CalculateDurations(v, B)
21:     end for
22:     N := UpdateLPN(N, V, B, Δ, T, Θ)
23:     N := InsertPseudoTransitions(N, V, B, Θ)
        return N
```

be either manually provided by the user or determined automatically via one of the two cost functions, located within the *GenerateThresholds* function. The two supported cost functions produce models whose data points are evenly distributed across the bins and bins whose rates span a minimal distance [43].

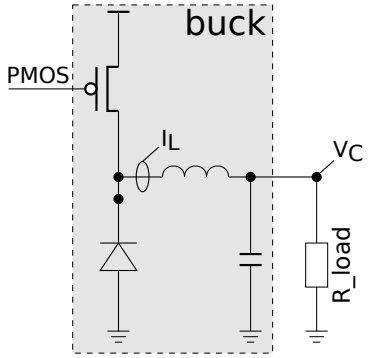


Figure 4.1: Analog part of a buck converter.

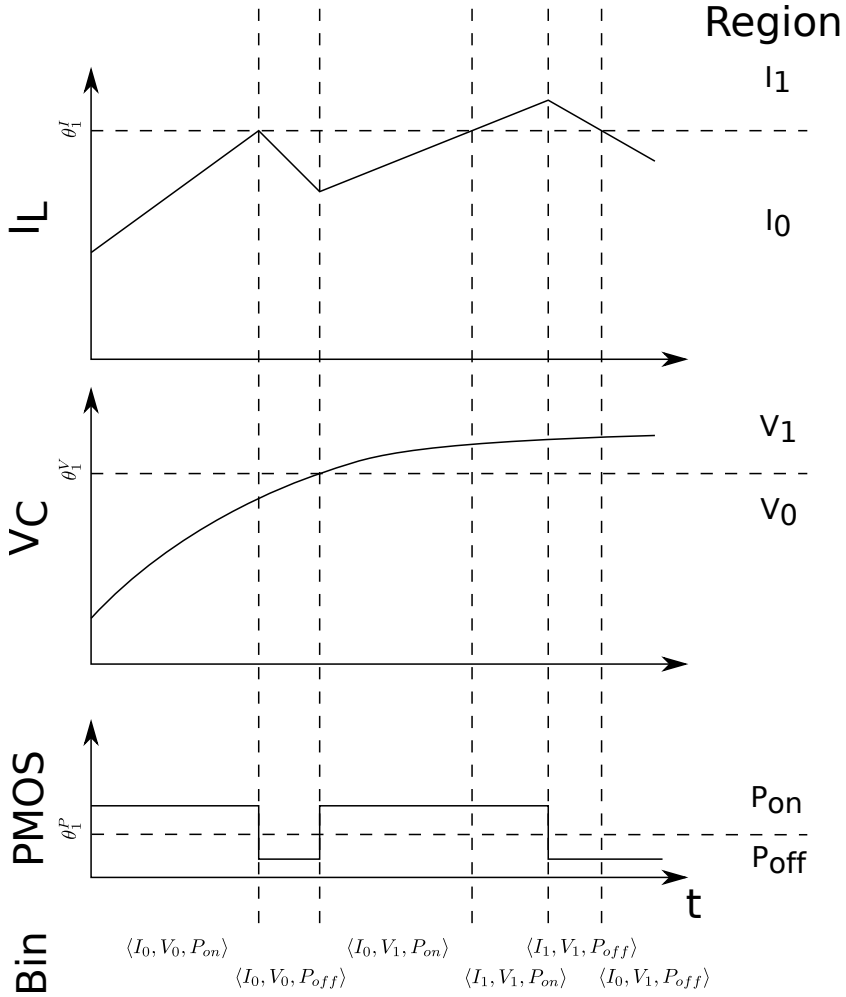
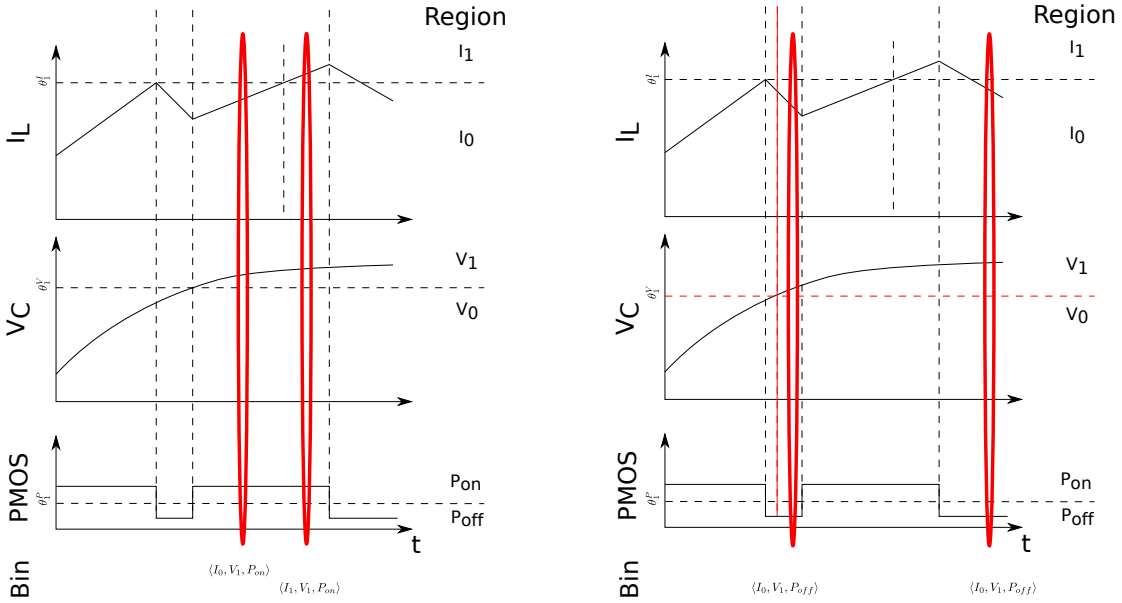


Figure 4.2: Threshold discretization.

To illustrate the process of the threshold based data discretization consider the analog part of the Buck converter, as shown in Figure 4.1. A portion of the simulation waveform, displayed in Figure 4.2, shows how the thresholds $\{\theta_1^I, \theta_1^V, \theta_1^P\}$ effectively group data points into distinct regions: $\{I_0, I_1, V_0, V_1, P_{off}, P_{on}\}$. A region, $\xi : \langle \theta_l, \theta_h \rangle$, is defined as a pair of thresholds θ_l



(a) Size increase.

(b) Precision loss.

Figure 4.3: Threshold discretization problems.

and θ_h[2]. For any given timestamp the superposition of these regions forms a unique state vector, called the *bin*. The data points are assigned to the corresponding bins by the *AssignBins* function [43].

The threshold based discretization method has a number of disadvantages. The model complexity and precision depend on the total number of thresholds and selecting quality thresholds can become a tedious and error prone task. While LEMA partly automates the process of picking optimal threshold the designer has to provide the total number of desired regions. This can potentially lead to the model size increase or loss of accuracy.

Furthermore, since regions are coupled together to form bins, selecting sub-optimal thresholds for one signal can affect other signals. For example, threshold based discretization of the inductor current in a buck converter leads to the data points from one charge cycle being assigned to several bins. The data points from the second charging cycle, while having the same rate, are being assigned to two different bins $\langle \mathrm{I}_0, \mathrm{V}_1, \mathrm{P}_{on} \rangle$ and $\langle \mathrm{I}_1, \mathrm{V}_1, \mathrm{P}_{on} \rangle$, as shown in Figure 4.3a. This results in the model size increase. On the other hand, if thresholds are not placed correctly, as illustrated in Figure 4.3b, data points with different rates might end up in the same bin, $\langle \mathrm{I}_0, \mathrm{V}_1, \mathrm{P}_{off} \rangle$, which leads to wide ranges of rates and decreases model accuracy.

[2]Low and high thresholds can be specified as negative and positive infinity respectively.

4.1.2 *Detecting discrete multivalued variables*

To reduce the potential space-state of the resulting model, digital-like signals are detected and modeled discretely. A variable, representing a digital signal, is defined as a d*iscrete multi-valued* (DMV) variable. Intuitively, a DMV variable is characterized as a variable, which approximates the original signal, using a finite set of constant values.

The detection of the DMV variables is handled by the *DetectDMV* function [43]. The function iterates over data points of a signal, associated with the variable, to determine stable regions. A region is considered stable if two conditions are satisfied:

- All of the region data points are located within ε neighborhood of each other: $\forall \sigma_i, \sigma_j \in \sigma : |\sigma_i(\upsilon) - \sigma_j(\upsilon)| < \varepsilon$;
- Region duration exceeds minimal stable time τ_{min}: $(\sigma_{|\sigma|}(\tau) - \sigma_1(\tau)) \geq \tau_{min}$.

A variable is reported as a DMV variable if the sum of durations of all stable regions, located within the signal, exceeds a threshold specified as $r_{stable} * T$, where r_{stable} is the stability ratio and T is the total duration of the signal. The tuple $\langle \varepsilon, \tau_{min}, r_{stable} \rangle$ forms a subset of the parameters *par*, required by the *DetectDMV* function. The same parameters as well as DMV detection criteria are used by the *CalculateValues* function to calculate values of DMV variables in each bin [10].

The main issue with the described method is that it relies on absolute values to determine DMV variables. To visualize this problem consider the **PMOS** signal, introduced earlier, that experiences transient behavior as illustrated in Figure 4.4. Depending on the ε parameter the second charging cycle can be considered an unstable region resulting in PMOS being treated as an analog signal. However, increasing the ε bound may affect DMV analysis of other signals and lead to originally analog signals be identified as discrete-signals. Furthermore, similar problem exists with the τ_{min} parameter, which can either result in transient regions be treated as stable if the value is too low or stable regions considered as transient if the value is too high. Picking the optimal parameters may depend on the particular waveform and require the designer extensive understanding of the discretization algorithm, which can undermine the idea of the automated model generation.

4.1.3 *Calculating ranges of rates*

LPNs model analog signals in a piece-wise linear manner, approximating continuously varying signals with ranges of rates. After the bins have been assigned to each data point, the rates of

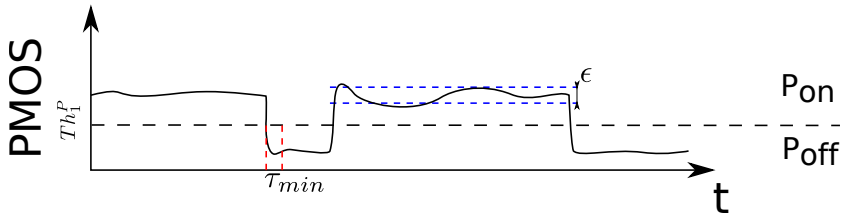


Figure 4.4: PMOS signal with transient.

all bins are calculated for each signal using the function *CalculateRates* [43]. A sliding window technique is used to smooth out transient pulses and other effects created by threshold edges. The size of the window, *ws*, is one of the parameters *par*, used in the model generation. Effectively, this method calculates the first derivative of every point assigned to a bin and reports maximum and minimum values of the derivatives as boundaries for bin rates.

4.1.4 *LPN synthesis*

Once input waveforms have been processed and raw data has been split into bins, the LPN construction process begins. Every bin is represented as a unique *place* in the generated model. Places are connected to each other via a set of *transitions* with enabling conditions representing either i) the threshold that is being crossed in the move of an analog signal from one bin to another or ii) the change in the value of a digital signal.

A summarized representation of the LPN synthesis process is given in Algorithm 4.2. The function *UpdateLPN* iterates over all data points of all signals to detect changes in bin assignments and update LPN *N* with additional places and transitions. Originally, the algorithm would add a new place for any individual bin, however that results in the generated models being too restrictive and able to produce only the behavior observed in the original waveform. To simplify the model and allow it to exhibit more general behavior the notion of *care variables* [39, 10] has been introduced. A new state in the model is created only when there has been a change in a variable, marked as a care variable.

If the current variable is marked as a care variable, the algorithm proceeds to find data points that have different bin assignments. The *AddPlace* function creates a place corresponding to the new bin and adds it to the LPN, provided the place a place with the given bin does not exist. The *GetEnablingCondition* function returns the enabling condition for a transition based on the bin assignments of the variables, marked as *input,* in its postset place. Finally, the *AddTransition* function creates a new transition if a transition does not exist between the given preset and postset places with the given enabling condition, and updates the transition with the new assignments if it already exists. The function uses values, Δ, and durations, T, calculated

by the *CalculateDurations* function, during data discretization step. Once all the signals have been processed the *MergeTransitions* function simplifies the LPN by merging transitions with similar enabling conditions and assignments.

Algorithm 4.2 LPN synthesis algorithm.

```
1:  function UPDATELPN(N, V, B, Δ, T, Θ)
2:      Input:
3:          N: existing LPN or nothing
4:          V: set of system variables
5:          B: a set of data bins
6:          Δ: a set of bin valus
7:          T: a set of bin durations
8:          Θ: a set of generated thresholds
9:      Output:
10:         N: updated LPN or new LPN
11:     //Iterate over system variables marked as care variable
12:     for all v ∈ V do
13:         if v ∈ V(Care) then
14:             b' := ∅
15:             //Iterate over variable data points and create a new place on bin change
16:             for all σ ∈ v do
17:                 //Get data point bin
18:                 b := B(σ)
19:                 if b' ≠ ∅ ∧ b ≠ b' then
20:                     N := AddPlace(N, V, b)
21:                     en := GetEnablingCondition(V(input), b, Θ)
22:                     N := AddTransition(N, b, b', en, Δ, T)
23:                 end if
24:                 b' := b
25:             end for
26:         end if
27:     end for
28:     N := MergeTransitions(N)
29:     return N
```

To prevent the model from reaching states of undefined behavior the model generator has the capability to add additional pseudo-regions [43] via the *InsertPseudoTransitions* functions, which sets rates of continuous variables to zero. This feature is used for modeling certain types of systems, such as voltage supply rails, in order to not exceed given system constraints. Another useful feature of the existing model generator is the ability to interpolate input signals [38]. As a result, it is possible to generate models, which can react to new input conditions, without the necessity to run additional simulations.

While the introduction of care variables and interpolation has helped, the model generator still has difficulties with producing more general models. For example, as demonstrated in Figure 4.5 all regions from the data bins are used in the LPN transition guard conditions to

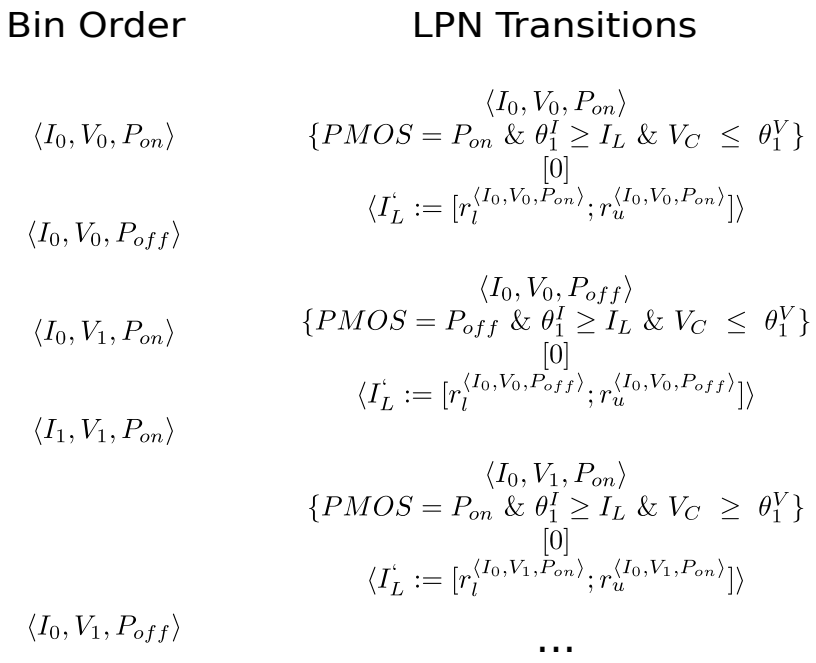


Figure 4.5: Bin to transitions translation.

determine the range of rates for the output signal. The bin $\langle I_0, V_0, P_{on}\rangle$ translates into transition with the same name, where every signal is checked against their respective thresholds. Such creation method of LPN transitions not only limits control over model fitness, as there are no means to specify how much information from the provided data set should be used in the model creation, but also might lead to resulting models being too restrictive.

4.2 ModelGen: proposed approach

The analysis of the model generation algorithm identified a number of problems related to the data discretization and LPN synthesis. To address these issues, a new algorithm has been developed [23]. A high-level flow diagram of the method is presented in Figure 4.6. The proposed approach seeks to improve the data binning by changing the discretization method as well as provide means of control over model fitness via the new algorithm, which has been called *rule mining*.

The flow starts with the discretization of the simulation data and the grouping of data points into discrete *states*. If necessary, the states can be filtered to reduce the amount of noise in the system. Afterwards, the states are analyzed to create a set of *data rules*. The data rules, which describe simulation *data patterns*, are used in the new LPN synthesis module.

The described flow is organized as a number of separate processing stages. While every stage operates and produces different data types, most stages share the same architecture. As

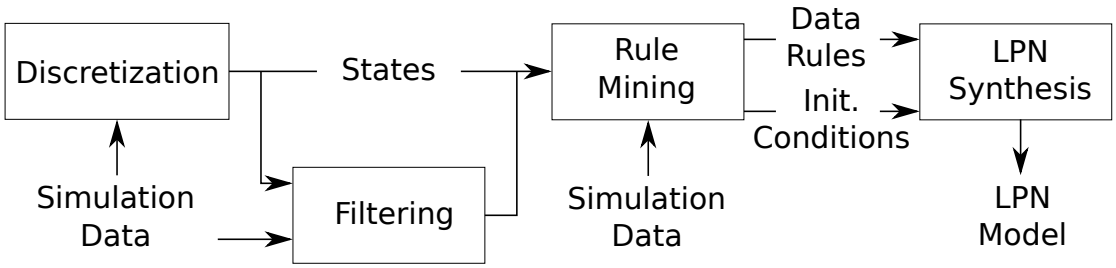


Figure 4.6: ModelGen flow.

described in Algorithm 4.3, a stage contains a set of *data processors*, which implement algorithms for processing input data, and a *data manager,* which governs the order in which the processors are applied. Additionally, every processor has a cost function to evaluate the processing cost of the data. For every element in the input data set, the main algorithm collects processing costs associated with the data processors of the stage, lines 5-10. It is possible that some of the processors are not applicable to certain data sets, which is indicated by their cost function returning a negative number. Afterwards, the data manager sorts the list of the processors based on their cost. The actual sorting order depends on the implementation of the *Sort*[3] function. The function *GetProcessor* returns the first processor of the sorted processor list, which is used to handle the data. If none of the processors can deal with the data, then the algorithm ends abruptly and reports an empty set, which should be treated as an error. The successfully processed data is grouped in the output data set.

The main motivation for creating such an architecture is to decouple algorithms, that process data, from algorithms that control data flow. The existing model generator has been designed in an ad hoc manner, which complicates introduction of new features. The new project aims not only to address issues, described in the previous sections, but also provide a convenient and modular framework for model generation. The addition of the cost function serves as a metric for comparing various algorithms and choosing the one that suits best for the current data set.

4.2.1 *Discretization*

The purpose of the discretization stage is similar to the data binning, that is grouping of simulation data points for further analysis by the LPN generation modules. However, in contrast to binning the proposed approach does not entangle data points from different signals. The new discretization methods operate on each signal individually and produce a number of discrete *states*. Formally, a state, s, can be defined as a tuple $\langle v, id, \tau_{start}, \tau_{end}, \Omega \rangle$ where:

[3]Unless specified otherwise, the default data manager sorts data in the ascending order.

Algorithm 4.3 Stage processing algorithm.

```
1:  function ProcessData(Input)
2:      Output := ∅
3:      for all in ∈ Input do
4:          ProcessCostSet := ∅
5:          for all processor ∈ DataProcessors do
6:              cost := processor.ProcessCost(in)
7:              if cost > 0 then
8:                  ProcessCostSet := ProcessCostSet ∪ {⟨processor, cost⟩}
9:              end if
10:         end for
11:         DataManager.Sort(ProcessCostSet)
12:         sortedProcessor := DataManager.GetProcessor()
13:         Out := sortedProcessor.ProcessData(in)
14:         if Out = ∅ then
15:             return ∅
16:         end if
17:         Output := Output ∪ {Out}
18:     end for
19:     return Output
```

- v is a system variable repsenting the original signal;
- id is a unique identification symbol;
- τ_{start} and τ_{end} are timestamps, indicating the beginning and end of the state respectively,
- Ω is a tuple, defining state values[4].

To better illustrate the difference between bins and states, consider Figure 4.7 as an example. The signal **PMOS** is represented as a sequence of states: $P^0_{on} \rightarrow P^1_{off} \rightarrow P^2_{on} \rightarrow P^3_{off}$, where the state name with the subscript, which represents a unique identification symbol, is analogous to the bin, while the superscript determines the state order.

Furthermore, every state contains additional parameters for approximating data points. Extra parameters depend on the type of approximation being used. For example, it is convenient to approximate the **PMOS** signal as a DMV variable with two boundary values, encapsulating all the data points from the range. Alternatively, the signal $\mathbf{I_L}$can be approximated in a piece-wise linear manner, which means that the state will contain information on ranges of rate.

In addition to the threshold based discretization, the new methods, based on the data clusterization [61], have been added. This has allowed the method to eliminate descretization problems, described in the previous sections, as well as provide finer control over model precision.

[4]This tuple depends on the state type. Most of the states define this tuple as $\langle b_l, b_h \rangle$, where b_l is the low boundary, such as a threshold or rate, and b_h is the upper boundary. Certain states, describing ideal digital signals, require only a single value.

The following sections detail algorithms used in the data processors of this stage. All of the discretization processors operate on a system variable $v : \langle Name, Type, \Sigma \rangle$, which contains a symbolic name, signal type: $\{input, output\}$, and data points from the simulation trace.

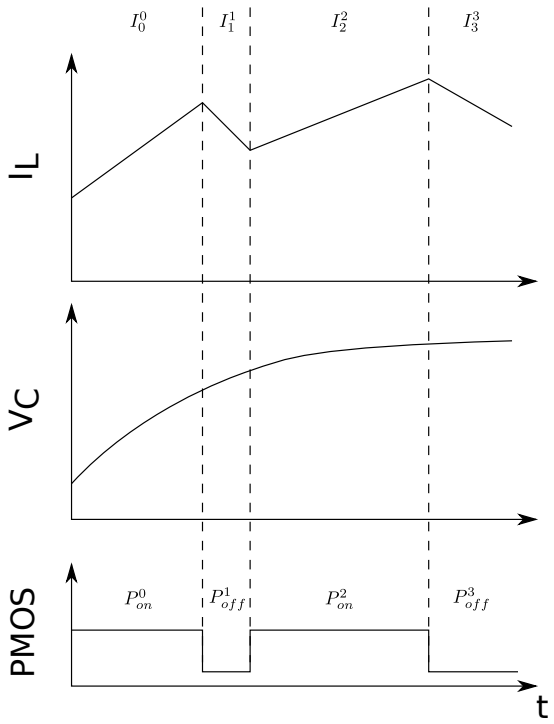


Figure 4.7: Derivative discretization.

4.2.1.1 Unique values discretization

The most simple and straightforward discretization method is based on grouping data points by their values. Certain data sets can have ideal discrete signals that do not contain any noise or transient, like **PMOS** signal in Figure 4.7. As a result it is possible to use computationally inexpensive discretization methods.

The algorithm is split between the module's cost and data processing functions and assumes that the cost function is used first to determine the possibility of applying the algorithm and the resulting processing price. The cost function, described in Algorithm 4.4, converts the input sequence of data points into a set and checks if the resulting set size exceeds a specified threshold of ***MAX_UNIQUE***. This parameter is needed to avoid applying this algorithm to analog signals, where every data point can have a unique value. In order to calculate the final cost, the function discretizes[5] the input data to obtain a sequence of states and uses it to generate

[5]While it is not shown in the cost function, all output results are cached in module's internal variables to minimize the amount of calculations.

a new signal. The function *GenerateSignal* iterates over all points of the input signal, finds the appropriate state according to the state and point timestamps, and creates a new data point, based on the state value. The generated signal is compared against the original one via the *CompareSignals* function, which measures the difference between signals as the sum of relative differences between every point in the original signal and the corresponding point with the same timestamp in the generated signal. The relative difference between the two points is calculated according to the following formula: $\frac{|P_{original} \cdot v - P_{generated} \cdot v|}{|P_{original} \cdot v|}$. The calculated difference multiplied by the module base cost is reported as the resulting cost of this data processor.

Algorithm 4.4 Unique values cost function.

```
1:  function ProcessCost(v)
2:      uniqueValues := ∅
3:      //Iterate over all data points of a variable and find the number of points with unique
        values
4:      for all point ∈ v.Σ do
5:          if point.v ∉ uniqueValues then
6:              uniqueValues := uniqueValues ∪ {point.v}
7:          end if
8:      end for
9:      //Check if number of unique states exceeds specified threshold
10:     if |uniqueValues| > MAX_UNIQUE then
11:         return −1
12:     end if
13:     states := ProcessData(v)
14:     generatedSignal := GenerateSignal(v.Σ, states)
15:     difference := CompareSignals(v.Σ, generatedSignal)
16:     return difference ∗ BASE_COST
```

If this algorithm is selected for the data discretization, then the data points are converted into the states according to the Algorithm 4.5. The method iterates over all points of the signal to detect sequences of points with the same value and creates new states via the *StateIdeal* function. This function[6] requires a system variable v, state timestamps τ_{start} and τ_{end}, and a state value. The produced states are grouped together in a sequence, which is returned as the result of this method.

4.2.1.2 Data values clusterization

The new version of the algorithm for handling DMV variables utilizes agglomerative clustering method [61] to circumvent the necessity for specifying multiple stability parameters by the designer. The data clusterization technique requires only *coefficient of variation* (CV) to au-

[6]Note, that state identification number *id* is generated by the function, using the provided state value.

Algorithm 4.5 Unique values process function.

```
1:  function PROCESSDATA(v)
2:      OutputStates := ∅
3:      σ_prev := v.Σ(1)
4:      //Find continous sequences of data points with the same value and form new states
5:      for all i ∈ [2, |v.Σ|] do
6:          σ_cur := v.Σ(i)
7:          if σ_cur.v = σ_prev.v ∧ i ≠ |v.Σ| then
8:              continue
9:          end if
10:         τ_start := σ_prev.τ
11:         τ_end := σ_cur.τ
12:         state := StateIdeal(v, τ_start, τ_end, σ_prev.v)
13:         OutputStates := OutputStates ∪ state
14:         σ_prev := σ_cur
15:     end for
16:     OutputStates := OutputStates ∪ StateIdeal(v, σ_prev.τ, σ_cur.τ, σ_cur.v)
17:     return OutputStates
```

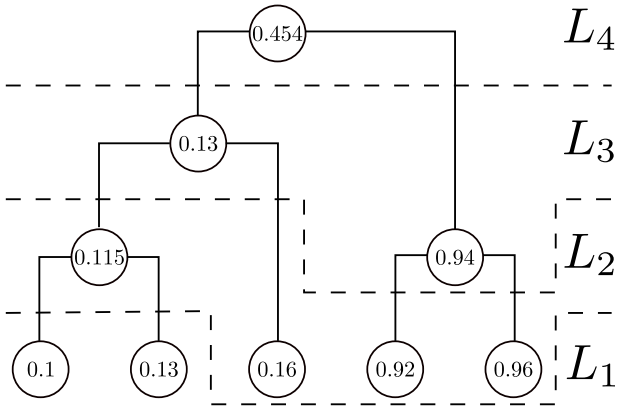


Figure 4.8: Dendrogram of data values.

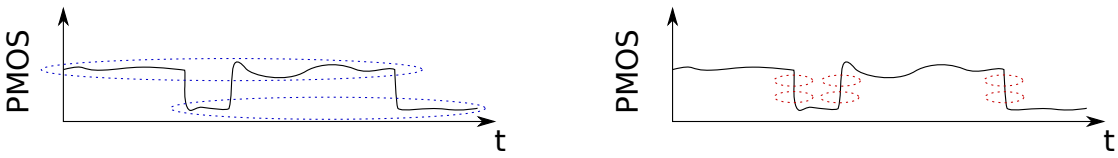
tomatically detect stable regions in data, which is a more universal characteristic for describing data, than a set of threshold constants.

In a similar manner, the operation of the algorithm is split between the cost and the data processing functions of the discretization module. The method, described in the Algorithm 4.6, starts by constructing a *dendrogram* from the provided data points and performing a layer search to determine a set of clusters, where each cluster has the relative standard deviation of data points below the specified CV parameter.

The illustration of this approach is provided in Figure 4.8. The creation of a dendrogram begins with transforming the input data points into *singletons,* clusters that contain only one point. At every step the function *FormDendrogram* iterates over all clusters to combine two

closest ones, according to the euclidean distance[7]. The initial singletons with values 0.1, 0.13, 0.16, 0.92, and 0.96 are analyzed to determine the pair with the closest distance to each other. This results in the first cluster with the value 0.115, connecting singletons 0.1 and 0.13, to be created. The newly created cluster and the rest of the singletons are analyzed again to find the best pair for merging. This iterative process repeats until only one cluster, called the *root*, is left.

Once the data is processed and the dendrogram is formed, a set of clusters is selected, where relative size of every cluster is below CV. Relative size of a cluster is calculated as $2|\frac{max-min}{max+min}|$, where *max* and *min* are maximum and minimum values located in a cluster. For example, the root cluster 0.454 has the relative size of 1.623 rounded up, which means that data variation for this cluster exceeds 100%. The *LayerSearch* function performs a breadth-first search on the dendrogram to find a layer, consisting of clusters conforming to the specified CV. Effectively, a layer is a set of clusters equidistant from the root cluster. As a result clusters 0.13 and 0.94 are located in the same layer L_3, while clusters 0.115 and 0.94 are located in the different layers L_2 and L_3, though being graphically present on the same horizontal level.



(a) Stable clusters.

(b) Transient clusters.

Figure 4.9: PMOS signal data clusterization.

The clusterization algorithm is applied to the discretization of a DMV signal. Figure 4.9a demonstrates how data points representing logical **0** and **1**, are grouped into two clusters of stable values. However, clusterization also creates clusters for transient points as shown in Figure 4.9b, which can lead to an excessive amount of states created. To cope with the transient regions the algorithm performs preliminary discretization of the data via the *CreateOutputStates* function. The *FindStableRegions* function relies on the adaptive low pass filter, described in Section 4.2.2.1, to remove transient states and related transient regions. If the total number of stable regions after filtration does not exceed the specified limit, the function proceeds by discretizing data once more to calculate the discretization error and report it as the metric cost of this method.

[7]Note that for better handling of ranges of rates negative and positive values are grouped into different clusters.

Algorithm 4.6 Data values clusterization cost function.

```
1: function PROCESSCOST(v)
2:     root := FormDendrogram(v.Σ)
3:     stabilityRegions := LayerSearch(root, CV)
4:     states := CreateOutputStates(v, v.Σ, stabilityRegions)
5:     stabilityRegions := FindStableRegions(states, stabilityRegions)
6:     if |stabilityRegions| > MAX_UNIQUE then
7:         return −1
8:     end if
9:     states := CreateOutputStates(v, stabilityRegions)
10:    generatedSignal := GenerateSignal(v.Σ, states)
11:    difference := CompareSignals(v.Σ, generatedSignal)
12:    return difference ∗ BASE_COST
```

If the cost function reports a positive value and the algorithm is selected by the data manager the module converts data points into states using values of stable regions, calculated earlier. The *ProcessData* function relies on *CreateOutputStates* function to produce the output states. This function, shown in Algorithm 4.7, iterates over all data points and picks a stable region with the value closest to the point. Using this data, the algorithm creates DMV states and puts them into an array. Finally, the output states are compressed via the *MergeEntries* function. This function finds states with the same *id*, that are located next to each other, and combines them to produce one state of longer duration. Note that transient points are also discretized, and it is the purpose of the filtering stage to detect and remove them.

4.2.1.3 Derivative clusterization

To enable finer control over ranges of rates for analog signals, similar clusterization technique is used. The data processing method from the previous section is adopted to group data points, based on their first derivative value (as opposed to fixed thresholds), as described in Algorithm 4.8.

Initially, the cost function checks if the data is marked as *input*, as this method can only be applied to the output variables. Afterwards, the *CalculateFirstDerivative* function calculates rate for every data point. These rates are subsequently partitioned by the clustering algorithm into a dendrogram and a suitable set of clusters is found, whose relative size conforms with the coefficient of variation. If the number of clusters exceeds the specified threshold the function report a negative number, indicating an error. Otherwise, the function returns the resulting cost of the operation in manner similar to previously described methods.

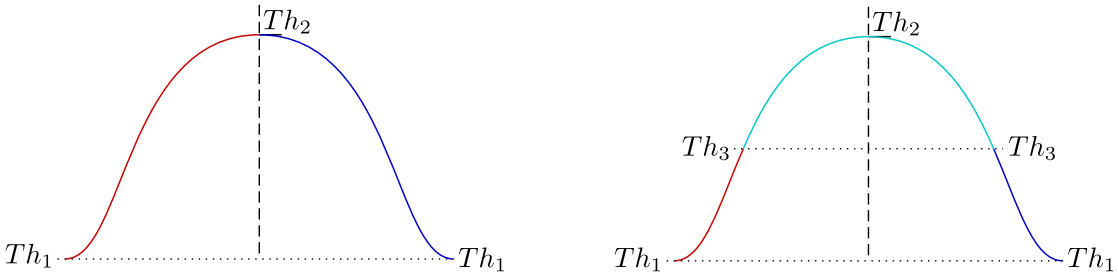
The operation of the data processing function is similar to that, described in the previous sec-

Algorithm 4.7 CreateOutputStates function.

```
1: function CREATEOUTPUTSTATES(v, data, stabilityRegions)
2:     Input:
3:         v: system variable
4:         data: data points to create states
5:         stabilityRegions: a set of data bins
6:     Output:
7:         outputStates: resulting data states
8:     outputStates := ∅
9:     //Iterate over all data points and create a new state for every point
10:    for all i ∈ [1, |data|] do
11:        //Find the closest cluster to the current data point
12:        cluster := GetClosestCluster(data(i), stabilityRegions)
13:        τ_start, τ_end := data.τ
14:        if i ≠ |data| then
15:            τ_end := data(i+1).τ
16:        end if
17:        state := StateDMV(v, τ_start, τ_end, cluster.min, cluster.max)
18:        outputStates := outputStates ∪ {state}
19:    end for
20:    //Merge similar states
21:    outputStates := MergeEntries(outputStates)
22:    return outputStates
```

Algorithm 4.8 Derivative clusterization cost function.

```
1: function PROCESSCOST(v)
2:     if v.Type = input then
3:         return −1
4:     end if
5:     derivData := CalculateFirstDerivative(v.Σ)
6:     root := FormDendrogram(derivData)
7:     stabilityRegions := LayerSearch(root, CV)
8:     if |stabilityRegions| > MAX_UNIQUE then
9:         return −1
10:    end if
11:    states := CreateOutputStates(v, derivData, stabilityRegions)
12:    generatedSignal := GenerateSignal(v.Σ, states)
13:    difference := CompareSignals(v.Σ, generatedSignal)
14:    return difference ∗ BASE_COST
```



(a) Sub-optimal thresholds.

(b) Adjusted thresholds.

Figure 4.10: Threshold discretization of a non-monotonic function.

tions, and reuses *CreateOutputStates* function, Algorithm 4.7, to produce output states. There are two notable distinctions. The distance to the cluster center, line 12, is calculated for the derivative instead of data point values:

$$distance := |region.center - derivData(i)|$$

Additionally, instead of a DMV state, line 17, this method produces states that approximate signals via ranges of rates, which are derived from the cluster's minimum and maximum values.

$$state := StateRange(v, \tau_{start}, \tau_{end}, cluster.min, cluster.max)$$

4.2.1.4 Threshold based discretization

One of the problems the threshold based discretization faces is determining discretization thresholds for non-monotonic functions. It is not possible to retrieve full information about a signal by just observing data points values and testing them against a set of thresholds. Part of the information is encoded within the point's rate, which is not possible to check directly using LPN's guard conditions. Furthermore, if thresholds are not selected carefully, it is possible to have a situation, shown in Figure 4.10a , where increasing and decreasing parts of the function are represented using the same thresholds, which essentially makes them indistinguishable from each other.

To address this issue, the new discretization method determines monotonic parts of the function as described by Algorithm 4.9. The cost function of the module assigns every data point a positive or a negative value, depending on the value of the first derivative. The *ProcessData* function in line 5 reuses the function from the "unique values" discretization module and the obtained states are filtered via the adaptive low pass filter. After that, the function iterates over the original data points to find sequences of data points, which correspond to the same state. These sequences are used to form a set of monotonic parts of the input signal and calculate the cost of the operation.

Algorithm 4.9 Threshold discretization cost function.

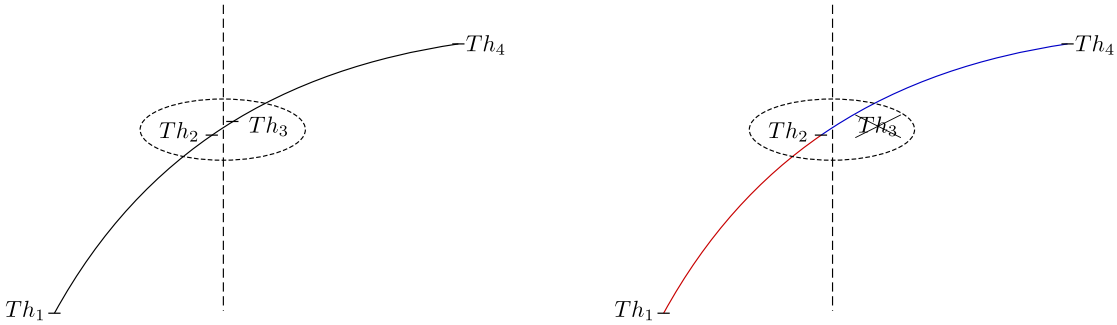
```
1:  function ProcessCost(v)
2:      //For every data point determine data sign
3:      dataSigns := AssignDataSign(v.Σ)
4:      //Create a collection of states
5:      signStates := ProcessData(dataSigns)
6:      //Remove noisy states
7:      signStates := FilterData(signStates)
8:      monotonicData, monotonicPart := ∅
9:      i := 1
10:     for all point ∈ v.Σ do
11:         if point.τ ∈ [signStates(i).τ_start, signStates(i).τ_end] then
12:             //Data points within the current sign state timestamps form monotonic part
13:             monotonicPart := monotonicPart ∪ {point.v}
14:         else
15:             //Add monotonic part to the collection of monotonic data
16:             monotonicData := monotonicData ∪ monotonicPart
17:             monotonicPart := ∅
18:             i := i + 1
19:         end if
20:     end for
21:     monotonicData := monotonicData ∪ monotonicPart
22:     //Store monotonic data internally to be used in Algorithm 4.10
23:     StoreMonotonicData(monotonicData)
24:     return |monotonicData| * NUM_GROUPS * BASE_COST
```

The set of monotonic pieces is stored internally in the module[8] and used by the data processing function, described in Algorithm 4.10. The function iterates over monotonic pieces and calculates thresholds via the *CalculateThresholds* function. This function distributes data points equally among the specified number of groups and calculates minimum and maximum value for each group as shown in Figure 4.11a. In this example data points are allocated between the two groups, which results in four thresholds being created. The obtained thresholds are then sorted in ascending order, and the thresholds that lie within the certain tolerance to each other are merged together by the *FilterThresholds*, as illustrated in Figure 4.11b. The algorithm proceeds by iterating over the data points to find the two thresholds that encapsulate the point value and creating states for every individual data point. After that the resulting states are compressed via the *MergeEntries* function, introduced earlier.

Algorithm 4.10 Threshold discretization process function.

```
1:  function PROCESSDATA(v)
2:      thresholds := ∅
3:      //Get monotonic data calculated in Algorithm 4.9
4:      monotonicData := GetMonotonicData()
5:      //Calculate discretization thresholds for monotonic data points
6:      for all monotonicPart ∈ monotonicData do
7:          partThresholds := CalculateThresholds(monotonicPart, NUM_GROUPS)
8:          thresholds := thresholds ∪ partThresholds
9:      end for
10:     //Merge close thresholds and sort in ascending order
11:     thresholds := FilterThresholds(thresholds)
12:     outputStates := ∅
13:     for all i ∈ [1, |v.Σ|] do
14:         τ_start, τ_end := v.Σ(i).τ
15:         if i ≠ |v.Σ| then
16:             τ_end := v.Σ(i+1).τ
17:         end if
18:         //Find two thresholds that encapsulate the current data point
19:         for all j ∈ [1, |thresholds| − 1] do
20:             if v.Σ(i) ≥ thresholds(i) ∧ v.Σ(i) ≤ thresholds(i+1) then
21:                 lowB := thresholds(j)
22:                 upB := thresholds(j+1)
23:                 state := StateThresholds(v, τ_start, τ_end, lowB, upB)
24:                 outputStates := outputStates ∪ {state}
25:                 break
26:             end if
27:         end for
28:     end for
29:     MergeEntries(outputStates)
30:     return OutputStates
```

[8]Transfer of monotonic data is handled by the *GetMonotonicData* function.



(a) Before merge.

(b) After merge.

Figure 4.11: Threshold discretization of a monotonic function.

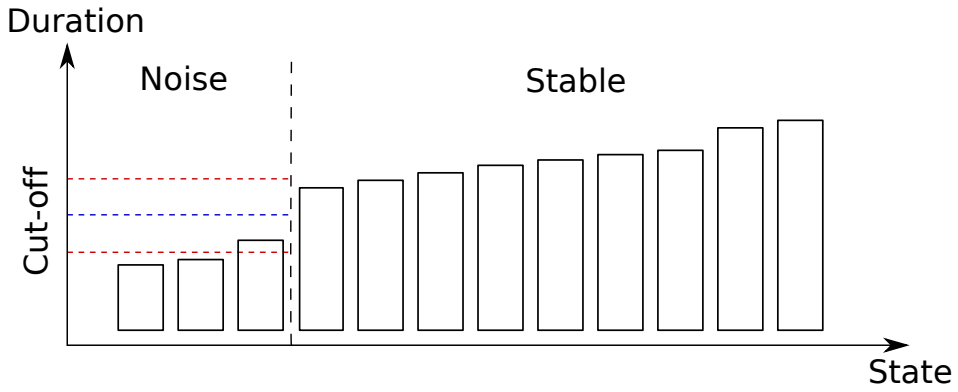
While this does not solve the problem with the non-monotonic functions completely, it allows for it to be handled by encoding signal behavior as a sequence of states. For example, applying this method to the waveform in Figure 4.10b introduces a new threshold Th_3, which transforms the waveform into a sequence of states: $S_1[Th_1, Th_3] \rightarrow S_2[Th_3, Th_2] \rightarrow S_1[Th_1, Th_3]$. As a result, the information about the signal behavior is encoded as sequences of states. It is important to note that this method is only used during the rule mining stage, when analog data is partitioned into smaller pieces. This provides a finer control over the number of thresholds produced, and, consequently, helps in reducing the model complexity.

4.2.2 *Filtering*

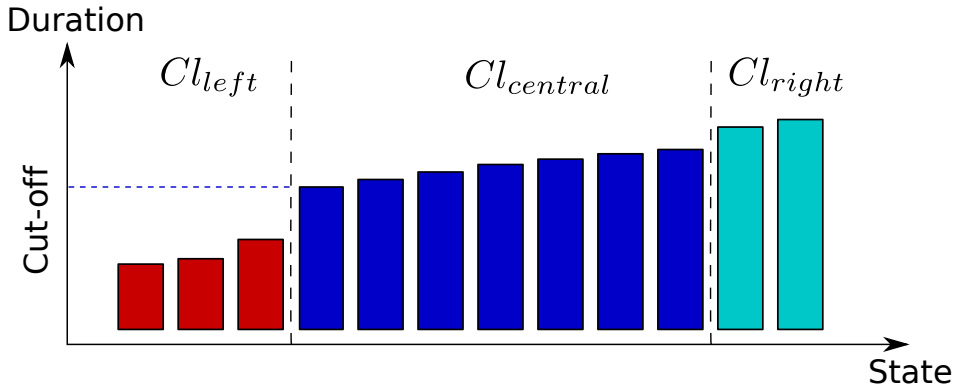
After the analog waveform has been transformed into a sequence of discrete states, it becomes possible to generate an LPN model of the system. However, such a model is likely to contain a lot of extra information due to "noisy" states. These relatively short-lived states obscure the important behavior of the system and are usually caused by the noise in the original analog signal. Thus, to reduce the model complexity and improve the state space analysis, filtering techniques are used in the model generator.

4.2.2.1 Adaptive low pass filter

Even a simple low pass filter can help to reduce the model complexity by removing small duration states. However, selecting the appropriate duration cut-off point can be a challenging task. An improperly chosen threshold can either result in noise states being present in the output or stable states being removed, as shown in Figure 4.12a. While it is possible to manually select the optimal threshold on a case-by-case basis this defeats the purpose of the process automation.



(a) Fixed cut-off duration.



(b) Adaptive cut-off duration.

Figure 4.12: State duration histogram.

The adaptive low pass filter attempts to automatically determine the optimal cut-off point. The cost function, described in Algorithm 4.11, serves as a metric function to indicate the efficiency and feasibility of the algorithm application. The function transforms input states into a sequence of state durations and calculates the total duration of the state trace. Afterwards the clusterization technique is used to group input states by their duration and find such a composition of clusters, where *relative duration* (RD) of any cluster, calculated as $\frac{Duration_{cluster}}{Duration_{total}}$, does not exceed the specified threshold. The obtained set of clusters is analyzed to determine the cluster with maximum total duration. The minimum duration in that cluster is used as the cut-off threshold to filter out input states.

The data clusterization naturally places noise states and stable states into different clusters, while the relative duration metric partitions clusters in such a way as to create a single cluster, which represents the majority of the stable states. Figure 4.12b illustrates how this approach produces three clusters with the cluster Cl_{center} containing most of the stable states. Choosing the leftmost state with the smallest duration in that cluster as the cut-off value preserves all of the states in that cluster as well as states with longer duration in Cl_{right}, while removing any

Algorithm 4.11 Low pass filter cost function.

1: **function** PROCESSCOST($states, v$)
2: $stateDurations := CalculateStateDurations(states)$
3: $totalDuration := \sum_{i=1}^{|stateDurations|} stateDurations(i)$
4: $root := FormDendrogram(stateDurations)$
5: $durationClusters := LayerSearch(root, totalDuration * RD)$
6: $clMaxDuration := GetMaxDurationCluster(durationClusters)$
7: $cutOffDuration := clMaxDuration.min$
8: //Remove states with duration below the threshold
9: $filteredStates := \emptyset$
10: **for all** $state \in states$ **do**
11: $duration := state.\tau_{end} - state.\tau_{start}$
12: **if** $duration \geq cutOffDuration$ **then**
13: $filteredStates := filteredStates \cup \{state\}$
14: **end if**
15: **end for**
16: $filteredStates := CorrectStates(states, filteredStates)$
17: $precisionLoss := PrecisionLoss(states, filteredStates, v)$
18: **return** $\frac{precisionLoss}{|states| - |filteredStates|}$

noise states from the cluster Cl_{left}.

After the noise states are removed, it is important to redistribute their duration between stable states and avoid any discontinuities in the state trace. The processing method of the filtering module relies on the *CorrectStates* function, explained in Algorithm 4.12. The algorithm iterates over the original input states to determine noise states that were removed. The missing states, located between stable states, are used to increase the duration of the current stable state. Finally, since it is possible that noise states separated two stable states with the same identification number, the *MergeEntries* function is used to merge any similar consecutive states.

The filtered states are used to generate a signal and compare it against the original signal, as demonstrated by Algorithm 4.13. The existing *CompareSignals* function is used to calculate the precision loss, incurred by the state removal. Note, that the simulation data is only used to provide a time reference, when generating a signal. The normalized precision loss per state removed is used to evaluate the method efficiency. The resulting states are stored internally in the module, so the data processing function does not perform any calculations and only retrieves the requested information.

4.2.2.2 Pattern based filter

While the low pass filter can improve the quality of the resulting models, it does not distinguish noisy states from short-duration states that are a meaningful part of the output signal.

Algorithm 4.12 Correct states function.

```
1:  function CORRECTSTATES(originalStates, filteredStates)
2:      noiseStates := ∅
3:      correctedStates := ∅
4:      index := 0
5:      for all state ∈ originalStates do
6:          correctedState := filteredStates(index)
7:          //If the state is stable its duration is increased by the duration of noise states
8:          if state = correctedState then
9:              IncreaseDuration(correctedState, noiseStates)
10:             correctedStates := correctedStates ∪ {state}
11:             index := index + 1
12:         else
13:             noiseStates := noiseStates ∪ {state}
14:         end if
15:     end for
16:     MergeEntries(correctedStates)
17:     return correctedStates
```

Algorithm 4.13 Precision loss.

```
1:  function PRECISIONLOSS(originalStates, filteredStates, v)
2:      signalOriginal := GenerateSignal(v.Σ, originalStates)
3:      signalFiltered := GenerateSignal(v.Σ, filteredStates)
4:      return CompareSignals(signalOriginal, generatedSignal)
```

For example, consider a sequence of states, shown in Figure 4.13a[9]. The sequence consists of the recurring state pattern: $\{S_1, S_2\}$, interrupted the by long duration state S_3. The duration histogram in Figure 4.13b illustrates that the adaptive low pass filter, described in the previous section, potentially can remove states S_2, that form part of the data pattern.

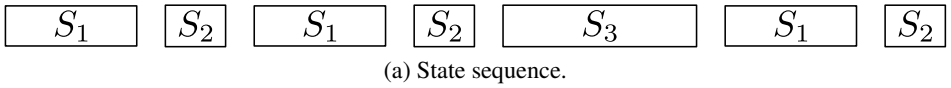
In order to overcome this problem, the pattern-based filter has been developed. The filter cost function, described in Algorithm 4.14, operates as a sliding window to determine the dominant state pattern in a window. The function calculates the window size as the percentage of the overall number of input states. For every state in the window, the function extracts state identification symbols, line 9. The set of these numbers essentially serves as an alphabet to create patterns via the *ExpandPatterns* function, Algorithm 4.15.

The *ExpandPatterns* function is effectively a function, which creates all possible words up to the size *PATTERN_SIZE*, from the given alphabet. The method starts by creating a set of patterns of size one, line 4. After that, the function expands the current set of patterns by adding additional state *id* to every pattern, line 14 and the newly created patterns are saved for the next

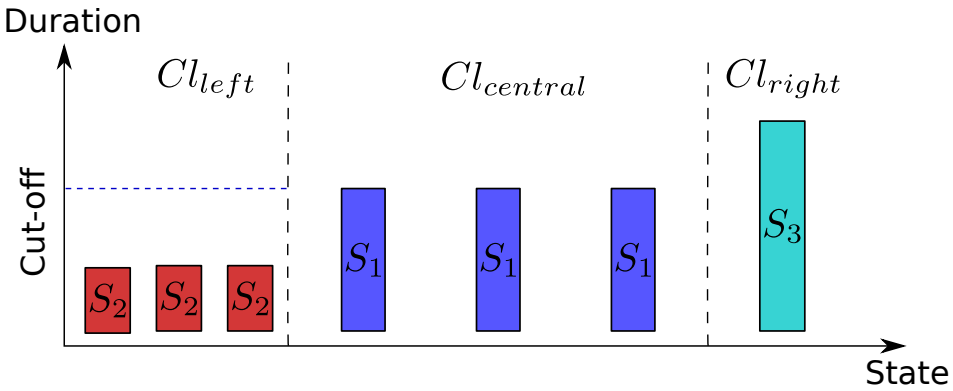
[9]The length of the rectangle represents state duration.

Algorithm 4.14 Pattern filter cost function.

```
1:  function ProcessCost(states, v)
2:      windowSize := ⌈|states| * WINDOW_RATIO⌉
3:      filteredStates := ∅
4:      for all i ∈ [1, |states|] do
5:          lowB := Max(0, i − windowSize/2)
6:          upB := Min(|states|, i + windowSize/2)
7:          //Extract states unique identification numbers
8:          stateIds := ∅
9:          for all j ∈ [lowB, upB] do
10:             if states(j).id ∉ stateIds then
11:                 stateIds := stateIds ∪ {state(j).id}
12:             end if
13:         end for
14:         //Create all possible state patterns
15:         windowPatterns := ExpandPatterns(stateIds)
16:         maxPattern := ∅
17:         maxWeight := −1
18:         //Determine dominant pattern in current window
19:         for all pattern ∈ windowPatterns do
20:             patternWeight := GetPatternWeight(pattern, states, lowB, upB)
21:             if patternWeight > maxWeight then
22:                 maxWeight := patternDuration
23:                 maxPattern := pattern
24:             end if
25:         end for
26:         //Preserve states that are part of the dominant pattern
27:         if states(i).id ∈ maxPattern then
28:             filteredStates := filteredStates ∪ state(i)
29:         end if
30:     end for
31:     filteredStates := CorrectStates(states, filteredStates)
32:     precisionLoss := PrecisionLoss(states, filteredStates, v)
33:     return precisionLoss / (|states| − |filteredStates|)
```



(a) State sequence.



(b) State histogram.

Figure 4.13: Low pass filter problem.

iteration. The function stops when the pattern size reaches the specified limit. For example, extracting state *id* numbers from Figure 4.13a yields alphabet: $\{1,2,3\}$. Given the pattern size limit as 2 the resulting patterns returned by this function are {"1", "2", "3", "12", "13", "21", "23", "31", "32"}[10].

Once all patterns are created the algorithm calculates weight for every pattern in the window, line 19, and chooses the pattern with the biggest weight. The function *GetPatternWeight*, explained in Algorithm 4.16, sums up duration of those states in the given window, that conform to the current pattern. The duration of the pattern in combination with the pattern size are used in the *Weight* metric function, which calculates the cost according to the following formula: $duration * BASE^{-size}$, where *BASE* is one of the parameters specified by the designer.

If the current state is part of the biggest pattern, it is preserved, otherwise it is removed from the state trace. The resulting cost of the method is also based on calculating the precision loss per state removed. The filtered states are stored internally and retrieved by the data processing function of the module.

4.2.3 *Rule mining*

Before the new model generator can construct an LPN model, it has to determine how much of the information, maintained within the original data set, is necessary to reliably describe the behavior of the system. Furthermore, due to expressiveness of LPNs, there are various possible methods to define the relationship between input and output signals. The proposed approach to

[10]While it is not shown in the algorithm, this function also performs minimization of the resulting patterns. Patterns, consisting of single *id*, such as "11", are removed from the result.

Algorithm 4.15 Expand patterns function.

```
1:  function ExpandPatterns(stateIds)
2:      curPatterns := ∅
3:      //Create patterns of length 1 first
4:      for all id ∈ stateIds do
5:          curPatterns := curPatterns ∪ {⟨id⟩}
6:      end for
7:      allPatterns := curPatterns
8:      size := 1
9:      while size ≤ PATTERN_SIZE do
10:         nextPatterns := ∅
11:         for all pattern ∈ curPatterns do
12:             //Expand existing patterns by adding state ids to every pattern
13:             for all id ∈ stateIds do
14:                 nextPatterns := nextPatterns ∪ {⟨pattern, id⟩}
15:             end for
16:         end for
17:         curPatterns := nextPatterns
18:         allPatterns := allPatterns ∪ curPatterns
19:         size := size + 1
20:     end while
21:     return allPatterns
```

Algorithm 4.16 Pattern weight function.

```
1:  function GetPatternWeight(pattern, states, lowB, upB)
2:      totalDuration := 0
3:      curId := 1
4:      prevId := 1
5:      for all state ∈ [states(lowB), states(upB)] do
6:          duration := state.τ_end − state.τ_start
7:          //Check if current state is part of the pattern
8:          if state.id = pattern(curId) then
9:              totalDuration := totalDuration + duration
10:             prevId := curId
11:             curId := (curId + 1) mod |pattern| + 1
12:             continue
13:         end if
14:         if state.id = pattern(prevId) then
15:             totalDuration := totalDuration + duration
16:         end if
17:     end for
18:     return Weight(|pattern|, totalDuration)
```

achieve this goal is by mining the *rules* - relations between the control state sequences and the output states[11]. The method of rule mining is designed to extract a minimal set of such rules that form a surjection from sequences of control states to the output states.

In contrast to the previously discussed processing stages the *Rule Mining* stage requires a few intermediate steps and overrides the stage *ProcessData* function, shown in Algorithm 4.3. As described in Algorithm 4.17, initially the stage processes a set of system variables V and a set of state sequences S to create a set of rules. After that, the algorithm iteratively detects conflicts between patterns and resolves them.

Algorithm 4.17 Data rule mining stage.

```
1:  function RULEMINING(V, S)
2:      Input:
3:          V: a set of system variables
4:          S: a set of state sequences
5:      Output:
6:          rules: a set of data rules
7:      rules := ExtractDataRules(V, S)
8:      while conflicts := DetectConflicts(rules) do
9:          rules := ResolveConflicts(conflicts)
10:     end while
11:     return rules
```

4.2.3.1 Extracting data rules

The extraction of the rules, as shown in Algorithm 4.18, begins with the processing of the simulation data to determine a set of signals that was not discretized. The function *GetNonDiscretizedData*, described in Algorithm 4.19, iterates over all system variables and probes the first state of every state trace to check, if the variable and the state name match. This serves as an indication that this signal was discretized. In case of a mismatch, the system variable is preserved and is used later to form data patterns.

A *data pattern (*DP*)* is a unique time-ordered sequence of all discrete input states that are contained between the start and end of the previous state. An additional part of a pattern is a set of data points from the continuous input signals. Effectively, a pattern samples and stores all information from input signals, which precede the changes of an output state. This information is interpreted as a triggering sequence to switch from one state to another. Formally, a pattern, P, can be defined as a tuple $\langle Out, States, Points, PreSet, PostSet \rangle$ where:

- *Out* - an output state,

[11] Control state is a state originated from a control signal. Similarly, an output state is a state originated from an output signal.

- *States* - a set of sequences of input states,
- *Points* - a set of sequences of input data points,
- *PreSet* - a set of patterns, which precede this state,
- *PostSet* - a set of patterns, which succeed this state.

Algorithm 4.18 Extract data rules function.

```
1:  function ExtractDataRules(V, S)
2:      dataRules := ∅
3:      initialConditions := ∅
4:      V_ND := GetNonDiscretizedData(V, S)
5:      for all outStates ∈ S do
6:          prevState := outStates(1)
7:          //Check if current sequence of states is marked as output
8:          if prevState.Type ≠ Output then
9:              continue
10:         end if
11:         initialConditions := initialConditions ∪ {prevState}
12:         //Iterate over output states to form data patterns
13:         for all i ∈ [2, |outStates|] do
14:             curState := outStates(i)
15:             patternStates := GetStatePatterns(S \ outStates, prevState)
16:             if CURRENT_STATE then
17:                 patternPoints := GetStatePoints(V_ND, curState)
18:             else
19:                 patternPoints := GetStatePoints(V_ND, prevState)
20:             end if
21:             dataPattern := ⟨curState, patternStates, patternPoints, ∅, ∅⟩
22:             rulePattern := ⟨curState, ∅, ∅, ∅, ∅⟩
23:             curRule := ⟨rulePattern, dataPattern⟩
24:             dataRules(curState.Id) := dataRules(curState.Id) ∪ {curRule}
25:             prevState := curState
26:         end for
27:     end for
28:     return ⟨initialConditions, dataRules⟩
```

In order to create the data patterns the method iterates over all state traces and picks those, that originated from signals marked as output, line 8. The first state in the trace is stored as initial condition, while the algorithm proceeds to iterate over all other states, including the output states, and extract the state patterns and data points for them. This is achieved via two functions: *GetStatePatterns*, Algorithm 4.21, and *GetStatePoints*, Algorithm 4.20. Both functions are similar in operation as they detect states and data points that reside within the time window of the specified state. While the pattern states always reside within the timestamps

of the previous output state, there is an option, governed by the *CURRENT STATE* parameter, to select the data points either for the current or the previous output state. Depending on the example the switch in the output state can be caused be an analog signal reaching a region at the same time as the switch occurred. Alternatively, the switch is related to an analog signal going through a series of regions, preceding the change in the output state.

Algorithm 4.19 Get non-discretized data function.

```
1:  function GETNONDISCRETIZEDDATA(V, S)
2:      data := ∅
3:      for all v ∈ V do
4:          for all s ∈ S do
5:              if s(1).Name ≠ v.Name then
6:                  data := data ∪ {v}
7:              end if
8:          end for
9:      end for
10:     return data
```

While determining the appropriate data points is rather straightforward and requires one to check, if the point timestamp is in between the start and end timestamps of the output state, detecting the correct pattern states requires additional checks. A state is considered to be part of the pattern, if it either started or ended during the time frame of the output state, as described in line 5 of Algorithm 4.21. Consequently, it is possible to have the same state to be a part of different data patterns. Finally, the previous state is added to the pattern and the entire pattern is converted to set format.

Algorithm 4.20 Get state data points function.

```
1:  function GETSTATEPOINTS(V, out)
2:      patternPoints := ∅
3:      for all v ∈ V do
4:          points := ∅
5:          for all point ∈ v.Σ do
6:              if point.τ ≥ out.τ_start ∧ point.τ ≤ out.τ_end then
7:                  points := points ∪ {point}
8:              end if
9:          end for
10:         patternPoints := patternPoints ∪ ⟨v.Name, v.Type, points⟩
11:     end for
12:     return patternPoints
```

The set format is a convenient way of representing information about the system state. Contrary to a state trace, which only captures changes in states, a set of state contains information about all system variables and is analogous to a node in a state graph. This difference

is best illustrated with Figure 4.7. For the output state I_2^2 the corresponding trace of pattern states is $I_1^1 \rightarrow P_{off}^1 \rightarrow P_{on}^2$. Converting this trace to the set format yields the following result: $\left\{I_1^1, P_{off}^1\right\} \rightarrow \left\{I_1^1, P_{on}^2\right\}$.

A list of data patterns extracted for the discretized signal of the inductor current in Figure 4.7 is provided below:

- $I_1^1 : \left\{I_0^0, P_{on}^0\right\} \rightarrow \left\{I_0^0, P_{off}^1\right\}$,
- $I_2^2 : \left\{I_1^1, P_{off}^1\right\} \rightarrow \left\{I_1^1, P_{on}^2\right\}$,
- $I_3^3 : \left\{I_2^2, P_{on}^2\right\} \rightarrow \left\{I_2^2, P_{off}^3\right\}$.

Every pattern captures the evolution of the system in the bin space between the changes of the outputs. For example, the pattern for the output state I_2^2 captures the system originally being in the bin $\left\langle I_1^1, P_{off}^1\right\rangle$, followed by the bin $\left\langle I_1^1, P_{on}^2\right\rangle$, until the system finally reaches the bin $\left\langle I_2^2, P_{on}^2\right\rangle$. Note, that no pattern is created for the state I_0^0 , as this is the initial state of the system.

Algorithm 4.21 Get state patterns function.

```
1:  function GETSTATEPATTERNS(S, out)
2:      patternStates := ∅
3:      for all inStates ∈ S \ outStates do
4:          for all in ∈ inStates do
5:              if in.τ_start > out.τ_start ∧ in.τ_start ≤ out.τ_end ∨
                   in.τ_end > out.τ_start ∧ in.τ_end ≤ out.τ_end      then
6:                  patternStates := patternStates ∪ {in}
7:              end if
8:          end for
9:      end for
10:     patternStates := out ∪ patternStates
11:     patternStates := ConvertToSetFormat(patternStates)
12:     return patternStates
```

Once the pattern data points and states are extracted a data pattern is formed, as show in line 21 of Algorithm 4.18. Additionally, a *rule pattern* (RP) is created for the same output state, but without any data points or state patterns. The combination of all data and rule patterns leading to the output state with the same identification symbols forms a mapping function, called a *data rule* (DR). Thus, an initialized rule consists of an output state and a set of pattern pairs, where one pattern is called a DP, as it has been extracted from the simulation data, and the other one is a RP, which is later used for model construction.

4.2.3.2 Conflict detection

After the rule extraction, all rule patterns are empty and must be filled with the data from their complementary data patterns. The purpose of the next steps of the algorithm is to determine how much of the information within the data patterns is needed to make a combination of all rule patterns a surjective function. To achieve this, the algorithm iteratively detects conflicts between patterns and resolves them.

A conflict occurs when one of the patterns is a subsequence of another pattern. Thus, a conflict reports that one cannot determine which output state the input stimulus leads to. Detection of conflicts is a two step process:

- Every RP of a rule is compared against every RP of every other rule.
- If an RP is not in conflict with any other RP, it is compared against every DP of every other rule.

The detection of conflicts between patterns relies on comparing sequences of state sets. Thus, it is necessary to define first how set comparison for states works. Given two sets, X and Y there are several possible comparison outcomes[12]:

- Sets are *equal*, if both sets contain the same elements and have the same size. For example, sets $X : \left\{A, B, C\right\}$ and $Y : \left\{A, B, C\right\}$ are equal.
- Sets are *unique*, if both sets contain at least one element not present in another set. For example, sets $X : \left\{A, B, C\right\}$ and $Y : \left\{A, B, D\right\}$ are unique.
- One set is a *subset* of another set (with another set being the *superset*), if all of the elements of the smaller set are present in the larger set. For example, set $X : \left\{A, B\right\}$ is the subset of the set $Y : \left\{A, B, C\right\}$.

Sequences comparison produces the same results but uses a different set of rules. Specifically, for two sequences $\alpha : (X_i)_{i=1}^{N}$ and $\beta : (Y_j)_{j=1}^{M}$ the comparison rules are as follows:

- Sequences are *equal*, if $N = M, \forall i = j | Compare(X_i, Y_j) = equal$. For example, sequences $\alpha : \left\{A\right\} \rightarrow \left\{B\right\}$ and $\beta : \left\{A\right\} \rightarrow \left\{B\right\}$ are equal, while sequences $\alpha : \left\{B\right\} \rightarrow \left\{A\right\}$ and $\beta : \left\{A\right\} \rightarrow \left\{B\right\}$ are not.

[12] Comparison between set elements is done by comparing state *id*.

- Sequences are *unique*, if either $\exists i : \forall j \geqslant i | Compare(X_i, Y_j) = unique$ or $\exists j : \forall i \geqslant j | Compare(X_i, Y_j) = unique$. For example, sequences $\alpha : \left\{A\right\} \rightarrow \left\{B\right\} \rightarrow \left\{C\right\}$ and $\beta : \left\{A\right\} \rightarrow \left\{D\right\}$ are unique.

- Sequence α is a *subset* of sequence β, if both sequences are neither unique, nor equal, and either: $\exists i, \exists j \geqslant i | Compare(X_i, Y_j) = superset$ or $N < M, \forall i = j | Compare(X_i, Y_j) = equal$. For example, sequence $\alpha : \left\{A\right\} \rightarrow \left\{B\right\}$ is a subset of sequence $\beta : \left\{A\right\} \rightarrow \left\{B\right\} \rightarrow \left\{C\right\}$. Additionally, sequence $\alpha : \left\{A\right\} \rightarrow \left\{B\right\}$ is a subset of sequence $\beta : \left\{A, B\right\}$.

The function *DetectConflicts* of Algorithm 4.3 returns a list of rule patterns, whose state pattern was marked as equal or a subset. However, it is important to note that no comparison is performed between the rule patterns, which have a dependency on each other. Before the comparison by set sequences takes place the conflict detection algorithm checks, if another RP is present in the postset or preset of the current RP, and reports no conflict in that case.

Detected conflicts are filtered to remove duplicate reports for the same conflicting rule pattern. Conflict resolution relies on a number of methods, each being applied in an order, determined by the associated cost function of the method. As such, the function *ResolveConflicts* reuses existing stage architecture, explained in Algorithm 4.3, and every one of these methods has a metric function and a data processing function in a manner similar to discretization and filtering modules. The following sections cover a range of conflict resolution techniques.

4.2.3.3 Resolve by state set

This method attempts to solve a conflict by expanding existing state sets in a rule pattern with additional states. Since a rule pattern is always a subset of its data pattern, the algorithm only looks at the last set of states in RP and DP to create a set difference and pick a signal, which states will be used to "grow" the RP. Once the signal is chosen the algorithm adds an additional state to every state set in RP from their complementary state sets in DP. Furthermore, when choosing a signal the method will prioritize input signals over output one and thus has separate costs for adding states of input and output signals[13].

This approach is best illustrated with the following example. Consider a data rule for an output signal C, consisting of the data pattern with the pattern states $DP : \left\{A_0^0, B_0^0, C_0^0\right\}_0 \rightarrow \left\{A_1^1, B_0^0, C_0^0\right\}_1 \rightarrow \left\{A_0^2, B_0^0, C_0^0\right\}_2 \rightarrow \left\{A_0^2, B_1^1, C_0^0\right\}_3$, and a rule pattern with states $RP : \left\{A_0^2\right\}_2 \rightarrow \left\{A_0^2\right\}_3$. Applying this resolution method will add states B_0^0 and B_1^1 to sets 2 and 3 of the RP

[13]By default this method reports the lowest cost for adding input states and the highest cost (second to Resolve by rule sequence method) for adding output states.

respectively and result in $RP : \left\{A_0^2, B_0^0\right\}_2 \rightarrow \left\{A_0^2, B_1^1\right\}_3$. Using this method once again will transform the rule pattern into $RP : \left\{A_0^2, B_0^0, C_0^0\right\}_2 \rightarrow \left\{A_0^2, B_1^1, C_0^0\right\}_3$. After that, the application of this method is impossible since all possible states have been added.

After the rule extraction, every RP is empty, so conflicts are reported for every rule pattern. Application of this method to the list of extracted DP in Section 4.2.3.1 transforms empty RPs into the following state:

- $I_1^1 : \left\{P_{off}^1\right\}$,
- $I_2^2 : \left\{P_{on}^2\right\}$,
- $I_3^3 : \left\{P_{off}^3\right\}$.

4.2.3.4 Resolve by set sequence

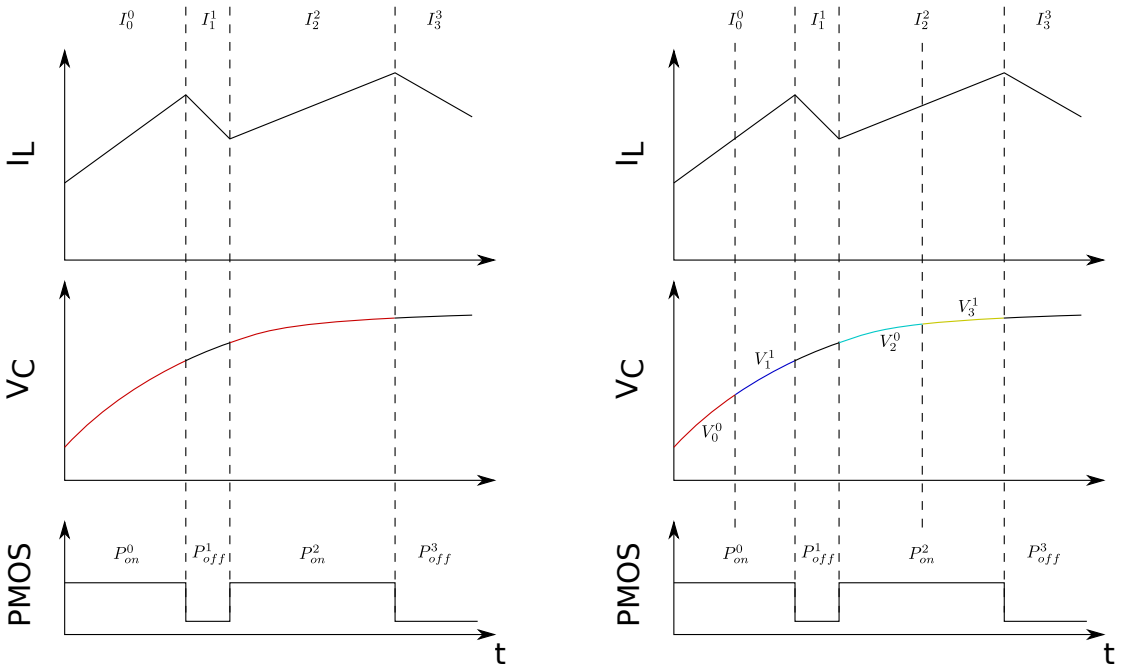
This resolution method adds additional state set sequences to a rule pattern in conflict. Note that the algorithm does not add new states, but detects states of signals that are already present in the sequence and adds states only for those signals. Only one set, preceding the last set in the rule pattern, is added at a time. The cost of the method is calculated as the multiplication of a base cost, set by the designer, and the resulting length of the pattern sequence. Additionally, this method cannot be used on an empty state pattern

Applying this method to $RP : \left\{A_0^2\right\}_2 \rightarrow \left\{A_0^2\right\}_3$, introduced in the previous section, transforms it into $RP : \left\{A_1^1\right\}_1 \rightarrow \left\{A_0^2\right\}_2 \rightarrow \left\{A_0^2\right\}_3$. This method can be used once more to expand the pattern and produce $RP : \left\{A_0^0\right\}_0 \rightarrow \left\{A_1^1\right\}_1 \rightarrow \left\{A_0^2\right\}_2 \rightarrow \left\{A_0^2\right\}_3$, which prevents further application of this algorithm.

The rule patterns described in the previous section contain a number of conflicts. Specifically, RP $I_1^1 : \left\{P_{off}^1\right\}$ and $I_3^3 : \left\{P_{off}^3\right\}$ are equal as both pattern contain the same output state P_{off}. Additionally, RP $I_2^2 : \left\{P_{on}^2\right\}$ is a subset of DP $I_1^1 : \left\{I_0^0, P_{on}^0\right\} \rightarrow \left\{I_0^0, P_{off}^1\right\}$ and $I_3^3 : \left\{I_2^2, P_{on}^2\right\} \rightarrow \left\{I_2^2, P_{off}^3\right\}$, which means that state P_{on} can lead to any of the output states. These conflicts can be partly solved by this method, which transforms RPs into the following sequences:

- $I_1^1 : \left\{P_{on}^0\right\} \rightarrow \left\{P_{off}^1\right\}$,
- $I_2^2 : \left\{P_{off}^1\right\} \rightarrow \left\{P_{on}^2\right\}$,
- $I_3^3 : \left\{P_{on}^2\right\} \rightarrow \left\{P_{off}^3\right\}$.

This effectively solves all possible conflicts for RP I_2^2 and makes it ready for LPN synthesis.



(a) Pattern data points.

(b) Additional states.

Figure 4.14: Dynamic data discretization.

4.2.3.5 Resolve by analog data

If there is any analog data available, the method discretizes the data by analyzing the set of analog data points, associated with the conflicting pattern. For the data points located within the timestamps of the output state only one additional state is created with thresholds calculated as minimum and maximum values found among the analog data. For the data points preceeding the output state the analysis algorithm uses the new threshold based discretization method, described in Section 4.2.1.4, to create additional states and insert them into the data pattern. Note, that the discretization does not affect other rules. Furthermore, since newly created states can increase the amount of state set sequences in DP, the RP is reset and addition of new states relies on previously discussed methods.

This method can be used to resolve conflicts between RP I_1^1 and I_3^3. As illustrated in Figure 4.14a, each of these patterns contains a set of data points. The threshold discretization algorithm can produce a variable number of states, depending on the provided settings. One possible outcome is shown in Figure 4.14b, where two additional states per rule are created, using the data points associated with the previous state. Note that extra states are added to non-conflicting rules only. These states change DP I_1^1 and I_3^3 and the resulting list of DP is given below:

– $I_1^1 : \left\{I_0^0, P_{on}^0, V_0^0\right\} \rightarrow \left\{I_0^0, P_{on}^0, V_1^1\right\} \rightarrow \left\{I_0^0, P_{off}^1, V_1^1\right\}$,

– $I_2^2 : \left\{I_1^1, P_{off}^1\right\} \rightarrow \left\{I_1^1, P_{on}^2\right\}$,

– $I_3^3 : \left\{I_2^2, P_{on}^2, V_2^0\right\} \rightarrow \left\{I_2^2, P_{on}^2, V_3^1\right\} \rightarrow \left\{I_2^2, P_{off}^3, V_3^1\right\}$.

Once the DP are expanded with new states, RP are reset and checked for conflicts once more. The resulting RP after conflict resolution appear as[14]:

– $I_1^1 : \left\{P_{off}^1, V_1^1\right\}$,

– $I_2^2 : \left\{P_{off}^1\right\} \rightarrow \left\{P_{on}^2\right\}$,

– $I_3^3 : \left\{P_{off}^3 V_3^1\right\}$.

At this point all conflicts have been resolved and the rule patterns are ready to be exported to the next stage.

4.2.3.6 Resolve by rule sequence

If the other methods fail, it is always possible to preserve the order of patterns, as they appear in the simulation trace. The algorithm, operating on pairs of data rules, determines their natural order via timestamps of the output state and adds the succeeding pattern to the postset of its predecessor and the preceding pattern to the preset of its successor. Normally, this method has the highest metric cost and ensures that conflict resolution cycle always ends.

The application of this method can be illustrated on a set of RP from Section 4.2.3.4. Assuming that no analog data is present to expand the state space, the only way to resolve the conflict between patterns I_1^1 and I_3^3 is by preserving their order. The rules preset and postset are updated accordingly and the resulting RPs are presented below:

– $I_1^1 : \left\{P_{on}^0\right\} \rightarrow \left\{P_{off}^1\right\}, PreSet : \left\{\emptyset\right\}, PostSet : \left\{I_3^3\right\}$,

– $I_2^2 : \left\{P_{off}^1\right\} \rightarrow \left\{P_{on}^2\right\}, PreSet : \left\{\emptyset\right\}, PostSet : \left\{\emptyset\right\}$,

– $I_3^3 : \left\{P_{on}^2\right\} \rightarrow \left\{P_{off}^3\right\}, PreSet : \left\{I_1^1\right\}, PostSet : \left\{\emptyset\right\}$.

[14]Provided that "Resolve by state set" method picks P_{off} state first.

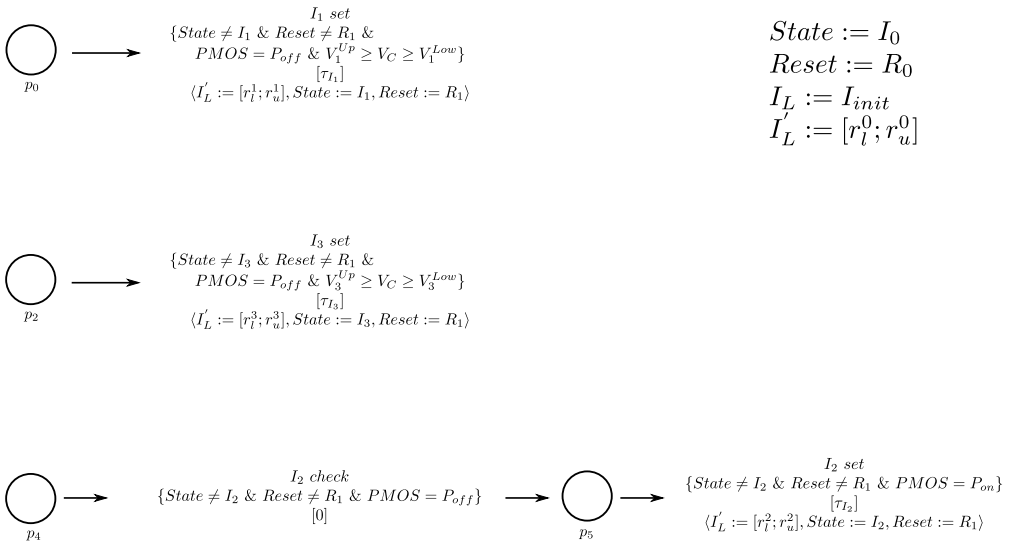


Figure 4.15: Partial model.

4.2.4 *LPN synthesis*

The new algorithm has been streamlined while keeping the main features of the existing model generator. The model generation process operates on the set of rules, extracted during the previous step, and starts with the elimination of redundant rules, that overlap with each other. Specifically, introduced conflict detection algorithm searches for rule pattern conflicts within every data rule and eliminates those, that have empty preset and postset and whose pattern was marked as equal or a subset. Note, that the eliminated rule updates the conflicting pattern with the additional timing information.

Every data rule is transformed into an LPN independently of others and resembles a state machine, which continuously checks input signals to determine, when to change the output state. This process is split between several important steps.

Initial conditions.

Initial conditions are formed from the set of output states, that occurred at the start of the state trace. Additionally, two extra variables, *State* and *Reset*, that govern the control flow of the model are added. It is important to note that no initial conditions are created for input signals and the generated model relies on other modules to properly set input variables.

Constructing partial models.

Every set of states in a sequence of state sets of a rule pattern is converted into a guard condition. These guard conditions are used to form a set of LPN transitions. The LPN transition, that originated from the last state set, is expanded with the delay information and assignment condition, based on the output state of the data rule. For every transition in the data rule a place

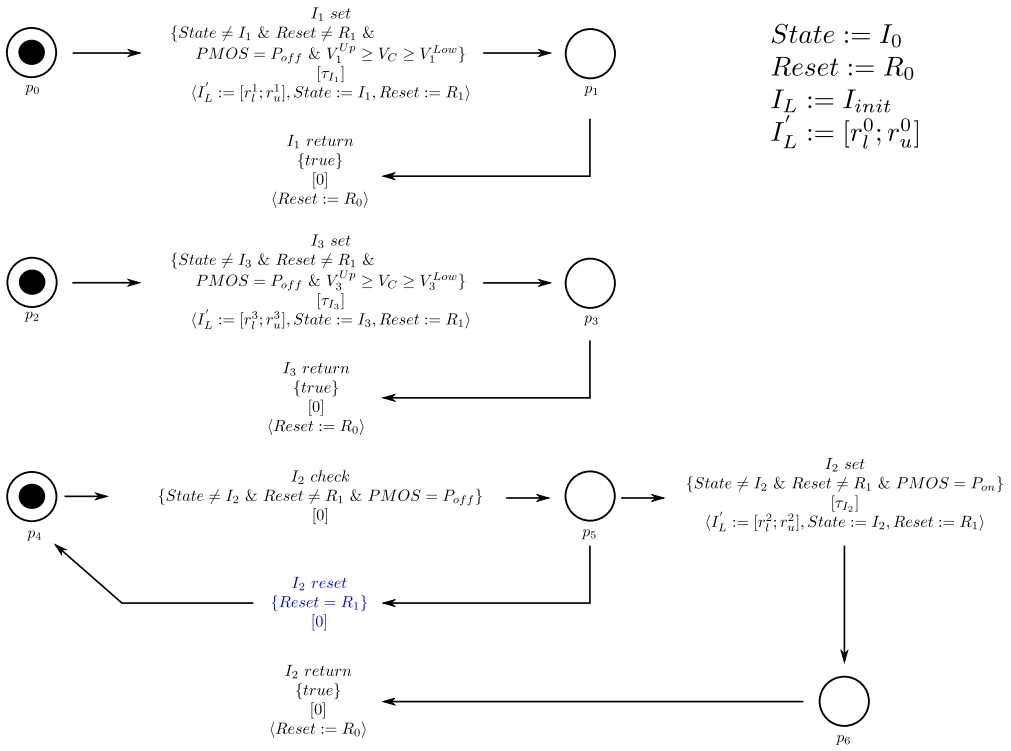


Figure 4.16: Partial model with reset links.

is added, which connects current transition with its predecessor.

The partial model, obtained from the set of data rules in Section 4.2.3.5, is shown in Figure 4.15. Transitions $I_1\,set$ and $I_3\,set$ for the data rules I_1^1 and I_3^3, respectively, have guard conditions that wait for the transistor to close and voltage over the capacitor to reach a specified threshold to set the lower and upper boundaries of the inductor current rate. Additionally, a special variable *State* is used to prevent models from reassigning output state continuously, thus live-locking the model. Data rule I_2^2, operating on only one input signal, is represented as a sequence of transitions, which effectively detects the rising edge of the PMOS signal. The delay value is calculated as the difference between the output state start timestamp and the start timestamp with the maximum value in the final state set. Note that delay information from the eliminated rules is used to create delay ranges. Since in this example every rule is unique only single delays are present.

Adding reset links.

The next step is the addition of reset links for all intermediate places in sequences of transitions, as illustrated in Figure 4.16. The purpose of the reset links is to revert the state of the detection sequence. For example, a token being present in place $p5$ means that the model has the potential to change the output state via transition $I_2\,set$. However, if transition $I_1\,set$ or $I_3\,set$ fires first it will invalidate any condition checks made by the sequence of transitions. The

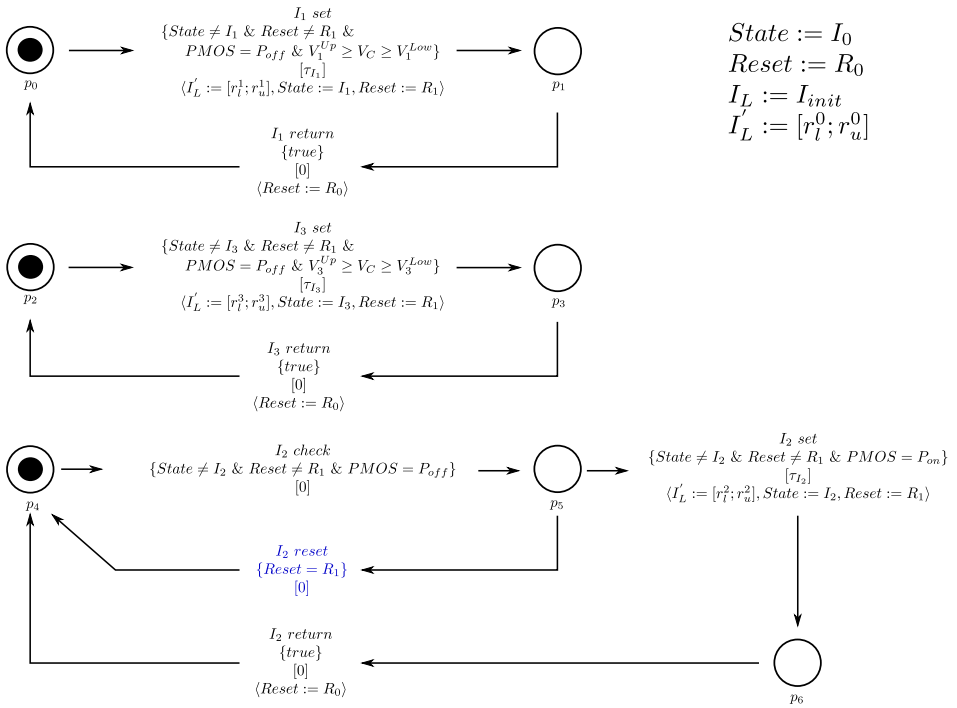


Figure 4.17: Complete model.

transition I_2 *reset* will move token back to the initial place $p4$ thus resetting the model. Furthermore, to avoid potential race conditions and concurrency between reset transitions and normal transitions, all reset transitions are marked as *persistent-enabled* transitions and have higher priority over other transitions. This effectively means, that if a reset transition is enabled, it will stay enabled regardless of signal changes, while the increased priority ensures that this type of transition will fire before any other enabled transition.

Additionally, the synthesis algorithm adds an intermediate place and a transition, which sets the reset variable to the the initial value, to every partial model. In essence, the *Reset* variable experiences a pulse, which indicates that the output state has changed and other partial models have to be reset. The *set* transition, which performs the state assignment, sets this variable to reset other parts of an LPN model. This approach as well as the ability of LPNs to transfer information between transitions via system variables help in minimizing the amount of connections in the model and improve the model analysis by the designer.

Connecting partial models.

Finally, partial models are connected, enclosed and initialized with tokens. If a data rule has empty postset and preset, then the synthesis algorithm connects the last transition to the first place in the model as shown in Figure 4.17, forming a loop. The resulting model of the system not only captures the behavior observed in the simulation trace, but it also is capable of

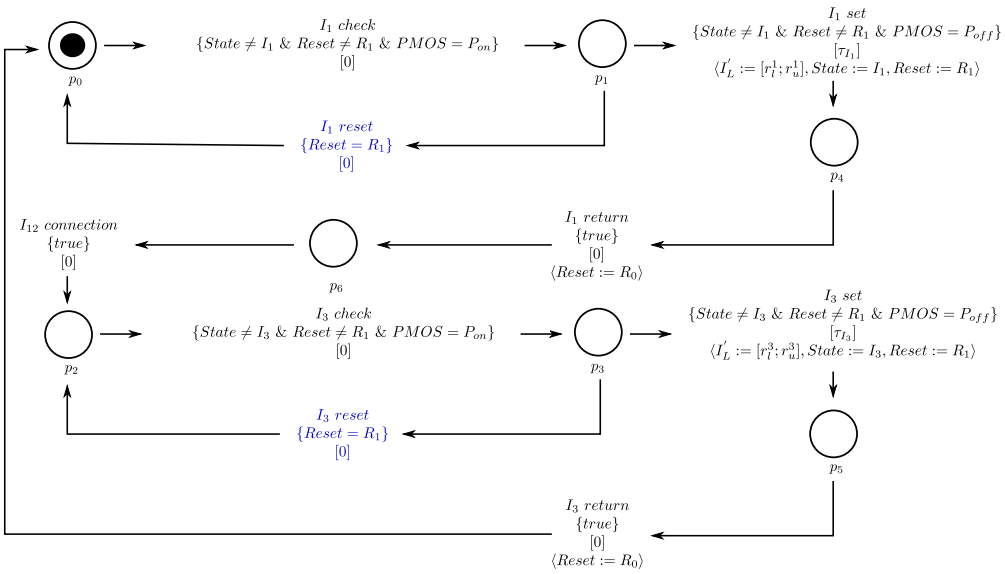


Figure 4.18: Connected model with rule dependency.

producing new behaviors thereby generalizing the simulation trace.

Alternatively, if a data rule has a dependency on another rule, the model generator connects the partial models according to their rule preset and postset conditions. First, for every rule in the postset a place is added to the partial model and connected to the last transition postset. If the rule preset is not empty, then a dummy transition without guard checks or assignments is created and connected to the preset of the partial model initial place. Afterwards, the new places are connected to the corresponding transition in another model as indicated by the data rule postset.

Finally, the combined partial models are enclosed to form a loop. If the model contains multiple transitions without a postset, then the algorithm connects additional places to every such transition. After that the transition serving as a connection between these places and all initial places without a preset is inserted into the model. If there is only one transition without a postset present, then no additional places are created and a connection is formed between this transition and all initial places.

The operation of this algorithm can be illustrated with a partial model, constructed for the data rules I_1^1 and I_3^3, obtained in Section 4.2.3.6. As shown in Figure 4.18 an additional place, $p6$, is added to partial model, derived from the rule I_1^1. At the same time a transition $I_{12}\,connection$ is added to the model for the data rule I_3^3. Combined together these elements form a connective bridge between partial models and ensure that only one model is active at a time. As a result the output state I_1, produced by the model, always precedes the output state I_3 in the same fashion as it was observed in the simulation trace.

4.3 Conclusion

Advantages of formal verification are offset by the difficulties in the generating of good system models. Furthermore, creation of abstract models requires substantial knowledge of formal methods. To address these issues a number of automated model generation techniques have been developed. The proposed flow presents an improvement to the existing method, which generates models from a set of simulation traces. This methodology seeks to provide a finer control over ranges of rates as well as an overall model structure. The following chapter covers a number of case studies and further illustrates the operation of the new model generator.

Chapter 5. Case Studies

The methodology described in the previous chapter was implemented as a stand-alone framework in Java. The tool utilizes a command line interface to generate an LPN from a single simulation trace. This chapter focuses on additional examples and explanation of the developed methodology. The first section covers a number of simple digital circuits to illustrate the operation of the rule mining. The subsequent sections detail the model generation process for the AMS systems, such as the C-element example, introduced in Chapter 3.

5.1 Digital circuits

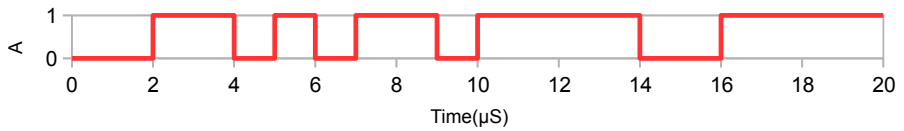
While the main focus of the model generator is directed towards the AMS systems the main idea behind the rule mining is related to describing digital circuits in one of the register transfer level (RTL) languages. The following examples show how the conflict resolution methods, such as *resolve by state set* or *resolve by set sequence*, can be used to create models for combinational or sequential circuits.

5.1.1 *Or element*

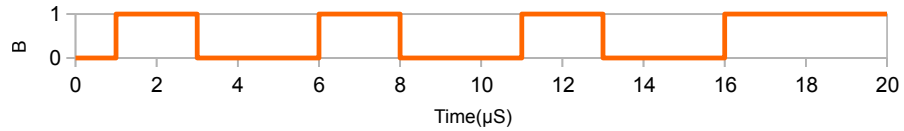
An OR element is a good example of a digital control module, which can illustrate how the rule mining can create abstract models for combinational circuits. The simulation waveforms, illustrated in Figure 5.1, were used in the model generation process. The signals A and B are inputs, while the signal C is the output. Note that the OR element reacts to the input changes without a delay.

All of the signals of the provided waveform were discretized with the *unique values* discretization method to produce a set of state traces. Due to the ideal nature of the waveform no filtering was needed and the obtained model is shown in Figure 5.2. The model is composed of 4 individual LPNs, expressing 4 possible combinations of the input signals. While the part of the guard condition that represents the input variables A and B is self-explanatory, the purpose of the *C_STATE* needs further clarification.

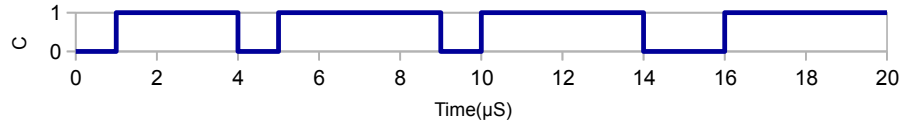
On the one hand this variable serves as a live-lock prevention mechanism. For example,



(a) Input A.



(b) Input B.



(c) Output C.

Figure 5.1: Or element waveforms.

when transition *T_C0* fires it changes the value of this variable to value, associated with the identification number[1] of the output state. Once the LPN has returned to the initial state this variable will invalidate the guard condition and prevent the transition from firing.

On the other hand the *C_STATE* variable is also used to resolve rule pattern conflicts during rule mining. The rule mining algorithm operates on the assumption that any information, located within a data pattern, is important and may lead to the output change. As a result an RP, while not in conflict with other RPs, might be in conflict with their respective DPs. For example, one of the rule patterns for the output state 0, $C_0^4 : \left\{A_0^4, B_0^3\right\}$[2], is in conflict with the data pattern $C_1^1 : \left\{A_0^0, B_0^0, C_0^0\right\} \rightarrow \left\{A_0^4, B_1^1, C_0^0\right\}$. To resolve this conflict the RP is expanded with the previous state $C_0^4 : \left\{A_0^4, B_0^3, C_1^1\right\}$, which is encoded as the *C_STATE* variable during the LPN synthesis process. While in this case it is possible to use the values of the output variable *C* directly there is no method to do that for analog signals. As such a separate variable is used to store the output state explicitly.

5.1.2 *Flip-flop*

In contrast to the combinational circuits sequential circuits, such as a flip-flop, require a different approach for model generation. Specifically, to capture the timing relation between input and output signals during the rule mining stage a higher priority is given to the *resolve by set sequence* method. The simulation waveforms, presented in Figure 5.3, are used to illustrate

[1]Note that the value of the identification number is different from the value of the C variable due to the usage of the hash function.

[2]The subscript represents the state identification number and is the same as the value of the variable. The superscript determines the state order and is derived from the state starting timestamp.

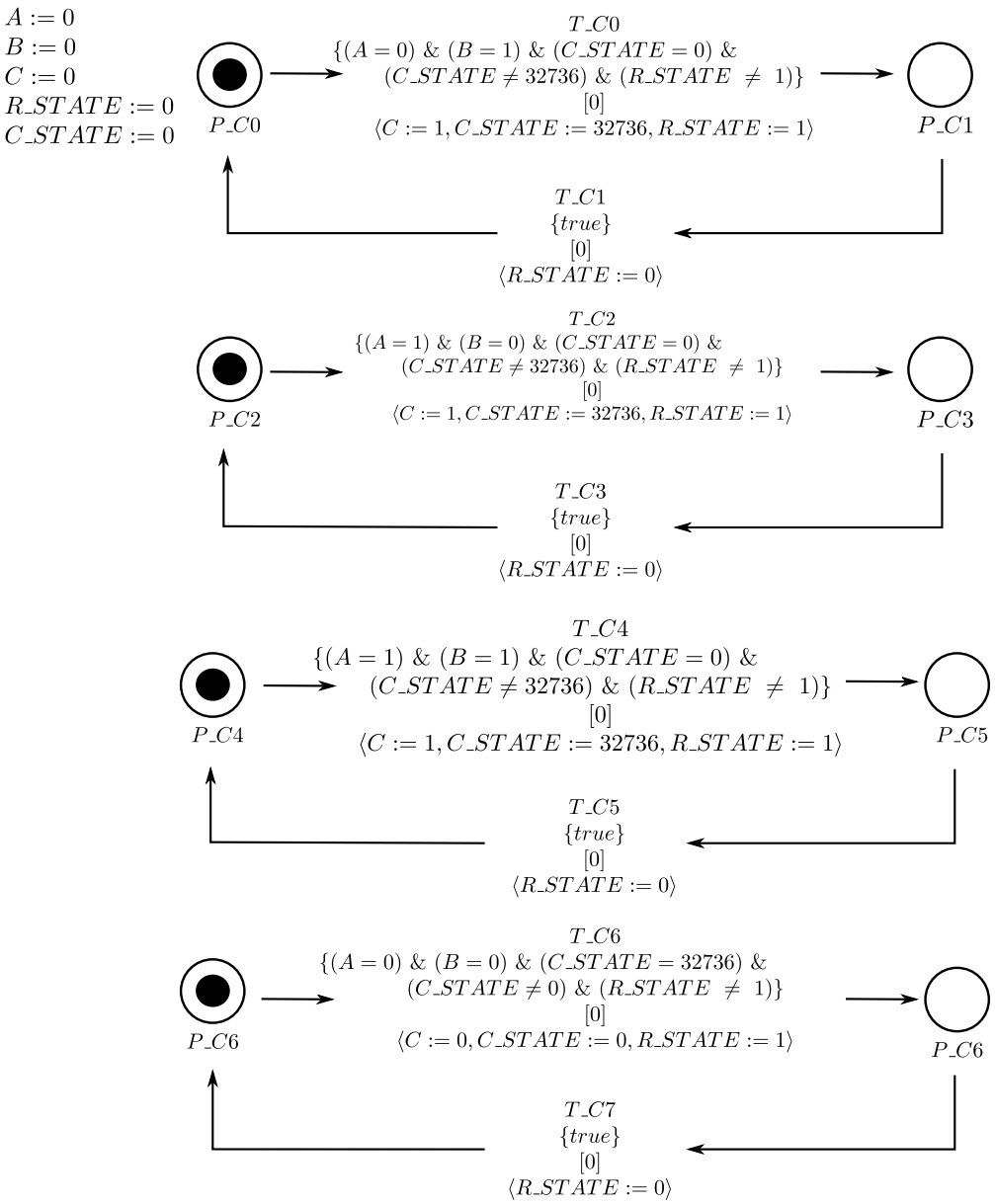
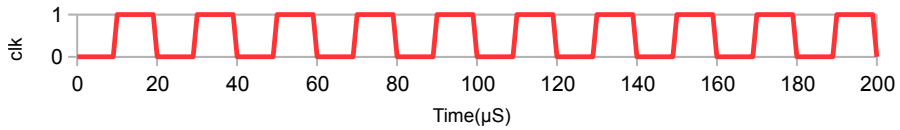
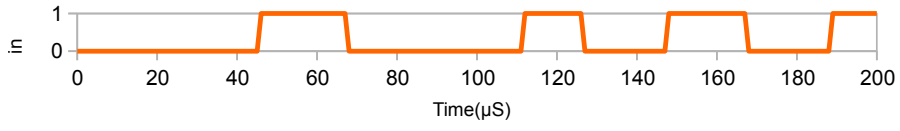


Figure 5.2: Or element model.

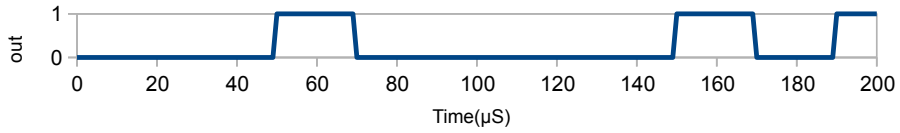
the process of model creation for the flip-flop circuit. The rising edge of the *clk* signal triggers the flip-flop to assign the value of the *in* signal to the *out* signal. The clock signal has a period of 20 time units, while the input signal alternates between 0 and 1 with a random delay between 5 and 50 time units.



(a) Clock.



(b) Input.



(c) Output.

Figure 5.3: Flip-flop waveforms.

Initially, the rule mining method attempts to resolve conflicts between empty RPs by adding the last state of the input signals from DPs. This results in the following RPs being created:

- $out_0^{70} : \left\{clk_1^{70}, in_0^{68}\right\}$,
- $out_0^{170} : \left\{clk_1^{170}, in_0^{168}\right\}$,
- $out_1^{50} : \left\{clk_1^{50}, in_1^{46}\right\}$,
- $out_1^{150} : \left\{clk_1^{150}, in_1^{148}\right\}$,
- $out_1^{190} : \left\{clk_1^{190}, in_1^{189}\right\}$.

While there are no conflicts between RPs themselves, all of the RPs have conflicts with DPs of another rule. For example, state set $\left\{clk_1, in_0\right\}$ is located within DP for the output state out_1^{50}. Unlike the previous example the output state is not used to resolve these conflicts as it is assigned a higher metric cost. Instead the *resolve by set sequence* algorithm is used and it transforms the RPs into:

- $out_0^{70} : \left\{clk_0^{60}, in_0^{68}\right\} \rightarrow \left\{clk_1^{70}, in_0^{68}\right\}$,
- $out_0^{170} : \left\{clk_0^{160}, in_0^{168}\right\} \rightarrow \left\{clk_1^{170}, in_0^{168}\right\}$,

– $out_1^{50} : \left\{clk_0^{40}, in_1^{46}\right\} \rightarrow \left\{clk_1^{50}, in_1^{46}\right\}$,

– $out_1^{150} : \left\{clk_0^{140}, in_1^{148}\right\} \rightarrow \left\{clk_1^{150}, in_1^{148}\right\}$,

– $out_1^{190} : \left\{clk_0^{180}, in_1^{189}\right\} \rightarrow \left\{clk_1^{190}, in_1^{189}\right\}$.

Although the RPs capture the behavior of the original circuit the RPs out_0^{70} and out_0^{170} still have a conflict with the DP of the rule out_1^{50}. In order to prevent the model generator from inserting additional state set sequences and reduce the resulting model complexity a dependency on the previous output state is added to resolve the conflicts. Which leads to the following RPs:

– $out_0^{70} : \left\{clk_0^{60}, in_0^{68}, out_1^{50}\right\} \rightarrow \left\{clk_1^{70}, in_0^{68}, out_1^{50}\right\}$,

– $out_0^{170} : \left\{clk_0^{160}, in_0^{168}, out_1^{150}\right\} \rightarrow \left\{clk_1^{170}, in_0^{168}, out_1^{150}\right\}$.

Finally, after all the conflicts are resolved the rules are minimized to remove the equivalent rules and the LPN model is generated, as shown in Figure 5.4. The reset transitions *T_out2* and *T_out6* play an important role in the model operation. Due to the random nature of the *in* signal it is possible to have tokens in the places *P_out1* and *P_out4*. While it is not possible for their post-set transitions to become enabled simultaneously it is possible for the model to produce incorrect behavior if the LPN is not reset after the output state change. For example, if no reset transitions are present transition *T_out1* can fire immediately after the input signal changes to 1 without checking the rising edge on the clock signal. In essence, reset transitions prevent the model from accumulating history from the previous pattern checks.

It is also important to note that dependency on the output in this case is excessive and can be safely removed from the final model. This dependency shows the importance of using good quality data that not only shows the desired behavior of the system but also does not contain or contains limited amount of undesired behaviors.

5.1.3 *Frequency divider with adder*

The final example in this section is the digital frequency divider. The divider, illustrated in Figure 5.5, consists of a single flip-flop, which feeds itself through the inverting output. The output of the divider and the original clock signal, connected through an inverter, are summed up via an adder. The resulting waveforms are presented in Figure 5.6.

The main purpose of this example is to demonstrate how the model generator constructs an LPN when insufficient amount of data to resolve conflicts is present in the data patterns. The clock signal, *f*, is selected as the input, while the output of the adder, *f_sum*, is selected as the

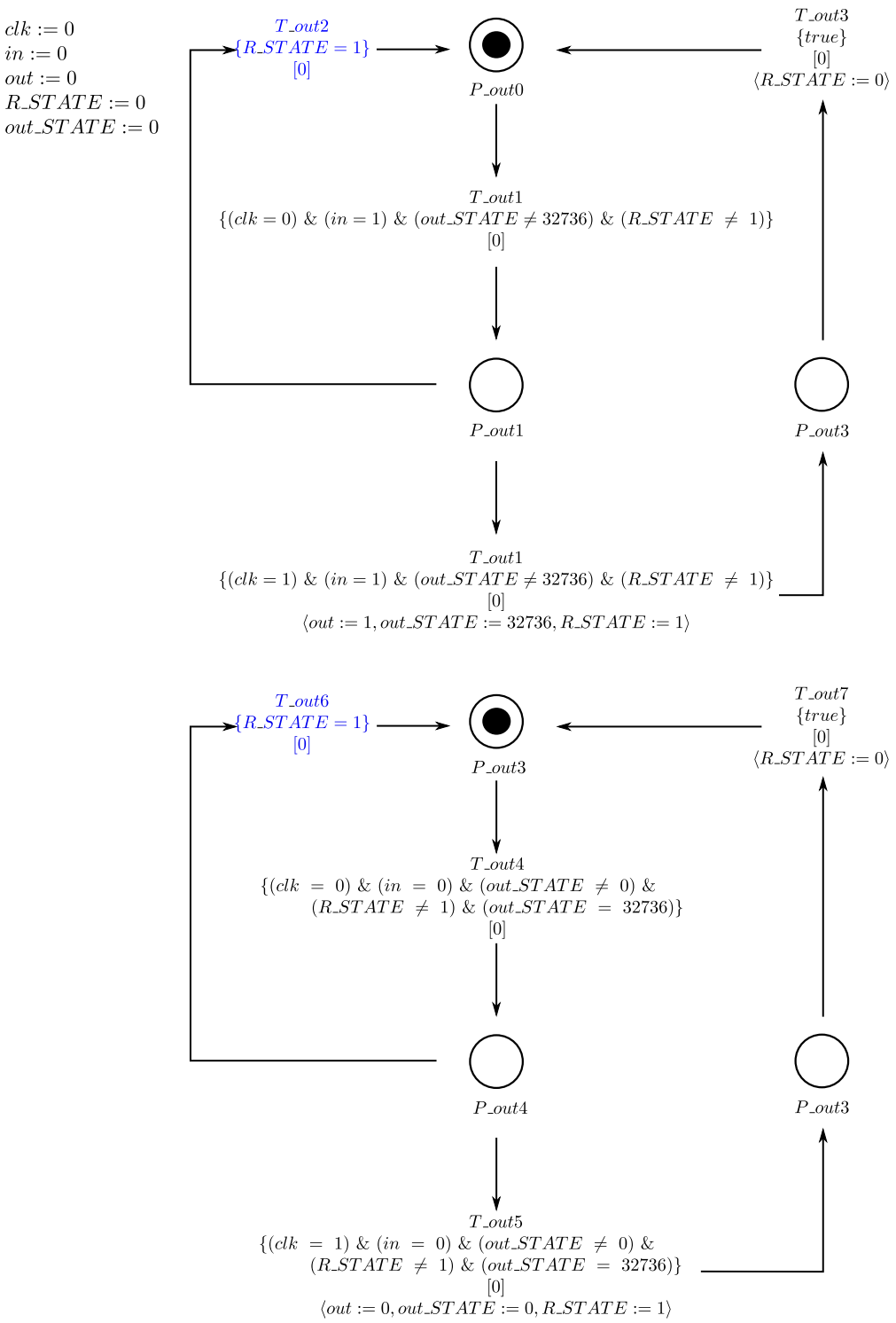


Figure 5.4: Flip-flop model.

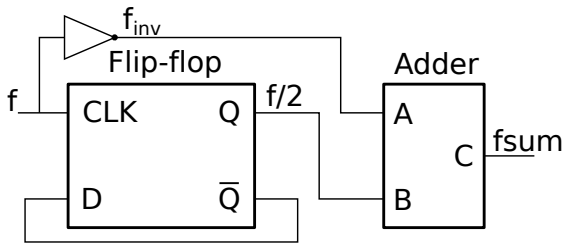
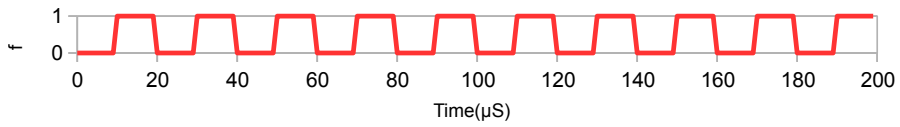
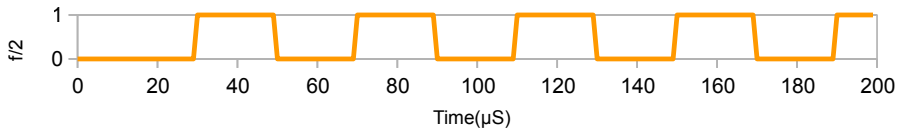


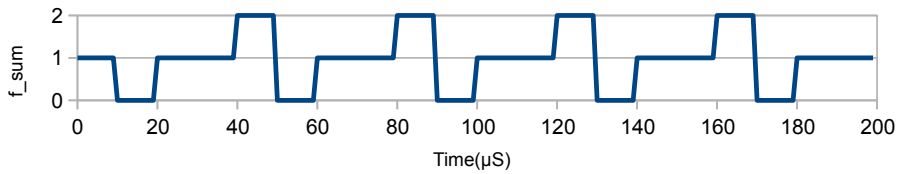
Figure 5.5: Frequency divider with adder.



(a) Clock.



(b) Clock/2.



(c) Clock sum.

Figure 5.6: Frequency divider waveforms.

output. In a manner similar to the flip-flop example, the initial set of conflicts is resolved by adding the final state of the input signal to every RP. Note that due to many RP being similar only RP with timestamps between 0 and 50 time units are listed below:

- $f\_sum_0^{10} : \left\{ f_1^{10} \right\}$,
- $f\_sum_0^{50} : \left\{ f_1^{50} \right\}$,
- $f\_sum_1^{20} : \left\{ f_0^{20} \right\}$,
- $f\_sum_2^{40} : \left\{ f_0^{40} \right\}$.

As states for both rising and falling edges are present in all DPs of these RPs a new set of conflicts is detected and the *resolve by state set* method is used to add a dependency on the previous output state to the RPs. This method is selected over the *resolve by set sequence* method due to having lower metric cost. This adjustment is necessary in this case as it helps to minimize the resulting complexity of the model. The update RPs are:

- $f\_sum_0^{10} : \left\{ f\_sum_1^{0}, f_1^{10} \right\}$,

- $f\_sum_0^{50} : \left\{ f\_sum_2^{50}, f_1^{50} \right\}$,
- $f\_sum_1^{20} : \left\{ f\_sum_0^{10}, f_0^{20} \right\}$,
- $f\_sum_2^{40} : \left\{ f\_sum_1^{20}, f_0^{40} \right\}$.

At this point only RPs $f\_sum_0^{10}$ and $f\_sum_2^{40}$ have conflicts as they are subsets of each others DP [3]:

- $f\_sum_0^{10} : \left\{ f\_sum_1^{0}, f_0^{0} \right\} \rightarrow \left\{ f\_sum_1^{0}, f_1^{10} \right\}$,
- $f\_sum_2^{40} : \left\{ f\_sum_1^{20}, f_0^{20} \right\} \rightarrow \left\{ f\_sum_1^{20}, f_1^{30} \right\} \rightarrow \left\{ f\_sum_1^{20}, f_0^{40} \right\}$.

The conflict resolution attempts to resolve these conflicts by extending the set sequences, however this does not help to solve this problem as the DP $f\_sum_0^{10}$ is a subset of the DP $f\_sum_2^{40}$. As a final resort the *resolve by rule sequence* method is used to form a causal dependency between these two rules, transforming RPs into the following list:

- $f\_sum_0^{10} : \left\{ f\_sum_1^{0}, f_0^{0} \right\} \rightarrow \left\{ f\_sum_1^{0}, f_1^{10} \right\}, PreSet : \left\{ \emptyset \right\}, PostSet : \left\{ f\_sum_2^{40} \right\}$,
- $f\_sum_0^{50} : \left\{ f\_sum_2^{50}, f_1^{50} \right\}$,
- $f\_sum_1^{20} : \left\{ f\_sum_0^{10}, f_0^{20} \right\}$,
- $f\_sum_2^{40} : \left\{ f\_sum_1^{20}, f_1^{30} \right\} \rightarrow \left\{ f\_sum_1^{20}, f_0^{40} \right\}, PreSet : \left\{ f\_sum_0^{10} \right\}, PostSet : \left\{ \emptyset \right\}$.

The resulting model is shown in Figure 5.7. The LPN models for the RPs $f\_sum_0^{50}$ and $f\_sum_1^{20}$ are constructed similarly to the previous examples. The RPs $f\_sum_0^{10}$ and $f\_sum_2^{40}$ are composed into a single LPN, which controls the order in which these states can appear in the state graph.

It is important to note that the obtained model does not correctly produce the behavior of the original system. The RP $f\_sum_0^{10}$ is an anomaly as it appears only once at the beginning of the simulation trace. However, the model generator does not have a mechanism to detect and handle such anomalies and treats them as a possible data pattern. This results in this pattern appearing in the simulation trace, presented in Figure 5.8, more than once. Although this can be viewed as a disadvantage of the new approach this example highlights the main principle of the model generator to create general models, which can produce new behavior.

[3] Conflicting parts are marked by blue and red colors.

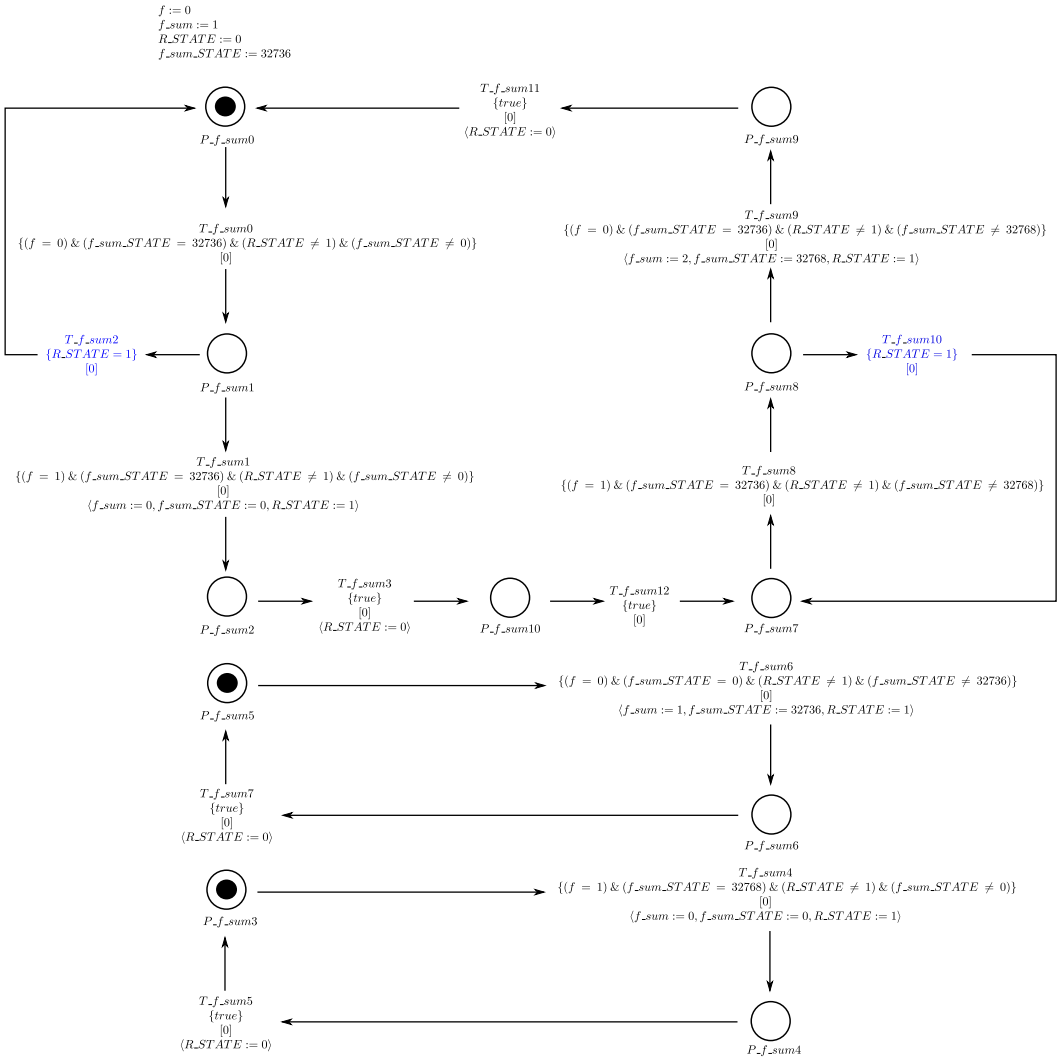


Figure 5.7: Frequency divider model.

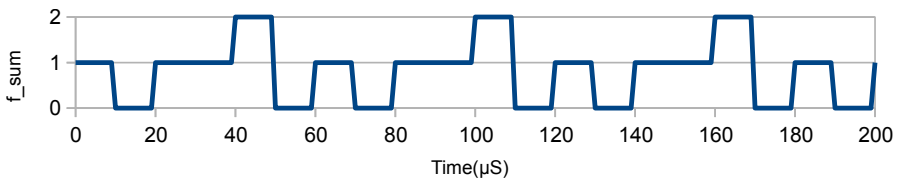


Figure 5.8: Frequency divider model waveform.

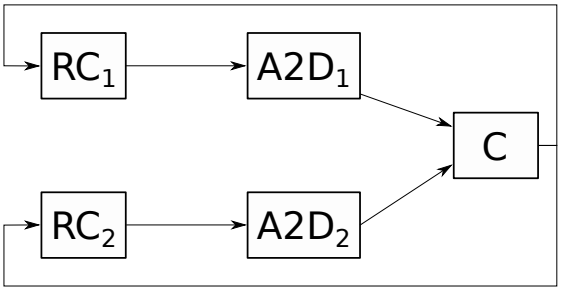


Figure 5.9: C-element modules.

5.2 C-element example

The C-element is a digital block widely used in asynchronous circuits. The C-element changes its value to logical zero, if both inputs are zero, and logical one, if both inputs are one. Otherwise it retains the previous value. While there are multiple implementations and specifications of this component the one used in this example is defined by the STG in Figure 3.1c in Chapter 3. This specification assumes that inputs do not flip randomly but hold their value between the output changes. Furthermore, the C-element used in this example has an inverting output, embedded into the specification. Although, it is possible to use a separate inverter, provided it has a negligible delay, it would also increase the model complexity. The output of the C-element is connected to the two RC circuits with different delay constants. Effectively, the RC circuits, acting as an analog environment for the C-element, force the inputs to change their value, once the capacitor is fully charged or discharged. The purpose of this example is to demonstrate model generator's ability to create LPN models for analog systems and show that the new approach is capable of producing more general models that can be reused.

An important step in constructing a system model is the decomposition of the model into individual components. While it is possible to create a model of the entire system, it might limit the reusability of the resulting model. As such every component is modeled individually, which allows to convert the original STG into the LPN format without any additional modifications. To connect the converted STG with the analog modules explicit A2D components are used, as shown in Figure 5.9. The simulation waveforms, presented in Figures 5.10 and 5.11, are used to create models for the RC circuits and the A2D converters.

5.2.1 *Analog to digital converter*

An analog to digital converter takes an analog input and discretizes it to be used by a digital component. Two A2D converter models for the inputs A and B are created separately by select-

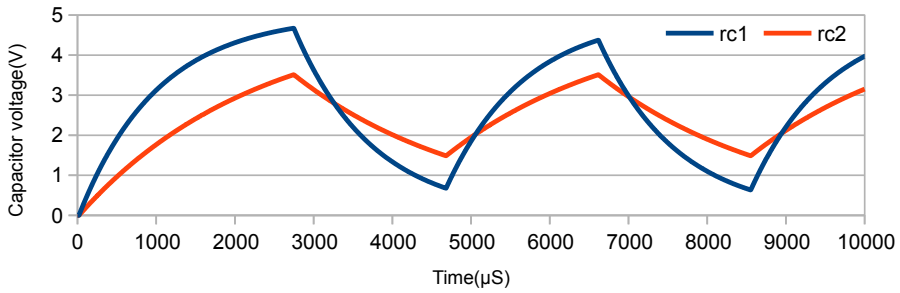
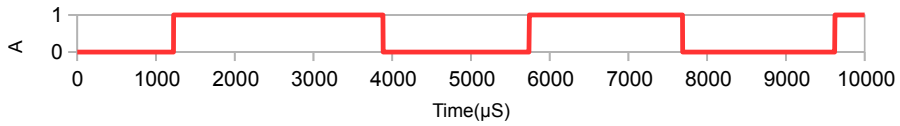
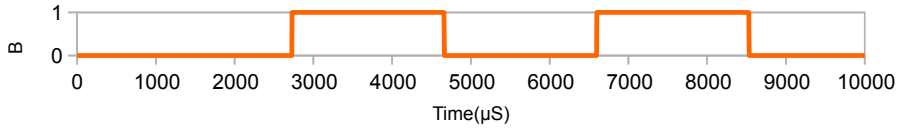


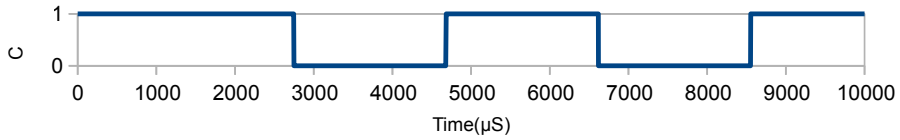
Figure 5.10: RC circuit waveform.



(a) Input A.



(b) Input B.



(c) Output C.

Figure 5.11: C-element waveform.

ing the variable *A* or *B* as the output and the variable *RC1* or *RC2* as the input respectively. The outputs, being ideal signals, are discretized with the *unique values* discretization method. The analog inputs are not discretized initially but instead partitioned into smaller bits and discretized during rule mining stage. As the output change is directly cause by the analog signal reaching threshold boundaries the *threshold based* discretization method is not used. Instead the data points within the time stamps of the output states are grouped together to produce a single state with thresholds selected as the minimum and maximum values.

The resulting model for one of the A2D converters is shown in Figure 5.12. The model is organized in a similar manner to the model of an OR element. The input variable *RC1* is used to determine the thresholds[4] for assigning the output.

[4]Note that the values are upscaled by a factor of 10000.

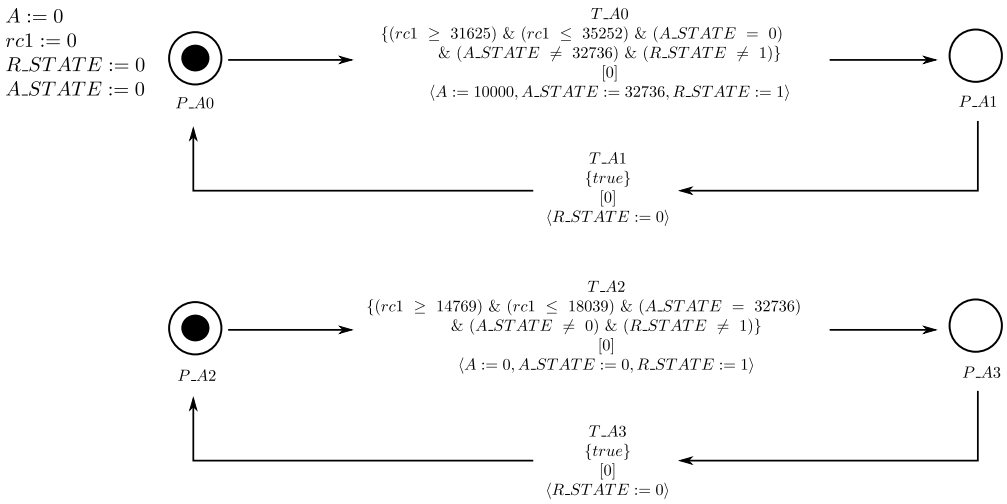


Figure 5.12: RC1 to A A2D converter.

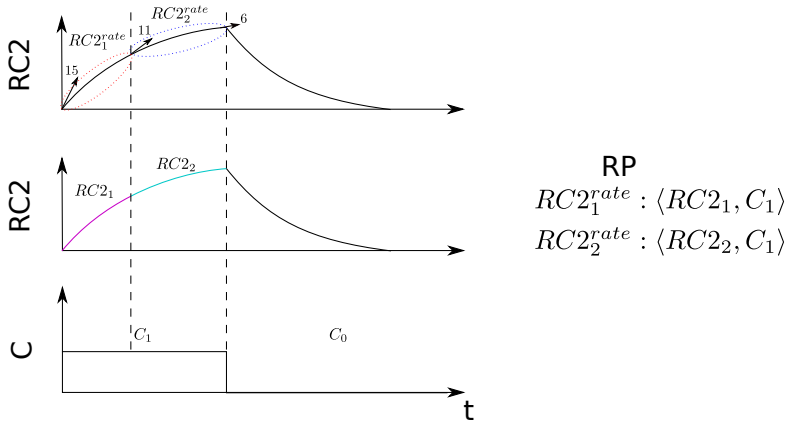


Figure 5.13: RC circuit binning.

5.2.2 *RC circuit*

An RC circuit is a simple example of an analog system. In order to create an efficient and general model of this system not only the output of the C-element is selected as input, but also the output of the RC circuit is selected as both input and output. This allows the model generator to use additional information of the analog signal to determine switching conditions.

The algorithm automatically determines that the C signal is best approximated as a DMV signal as it only has 2 distinct values. On the other hand the RC signal, set as the output is approximated via ranges of rates. For that the first derivative is calculated and clustering is used to group derivative values. For example, using the provided coefficient of variation the *RC2* is approximated via two states with ranges [11, 15] and [6, 11], as shown in Figure 5.13. Note, that the *RC2* signal set as input is discretized later on as the analog data is partitioned into smaller pieces.

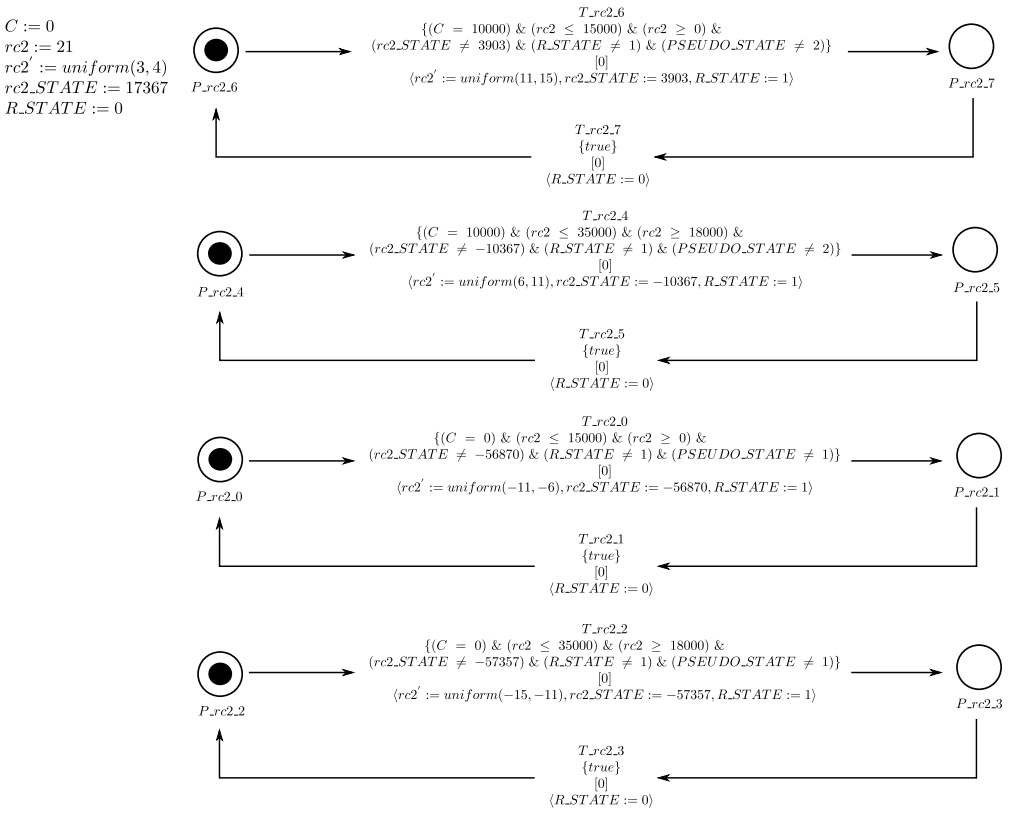


Figure 5.14: Improved RC2 model.

Once the data is discretized it is possible to form the rule patterns, which describe relations between input states and an output state. In this case the patterns for both output states $RC2_1^{rate}$ and $RC2_2^{rate}$ are identical and consist of the same state C1. As a result it is not possible to distinguish them and assign the output state properly, based only on observation of the signal C. To cope with that the analog signal *RC2* is discretized using a set of thresholds to produce additional pattern states $RC2_1$ and $RC2_2$. This expands the existing patterns and allows to distinguish the output states. In a similar manner, RPs for the decreasing part of the waveform are created.

The obtained rule patterns are transformed into an LPN, shown in Figure 5.14, and resemble a state machine, which continuously checks input signals to determine, when to change the output state. For example, transitions *T_rc2_4* and *T_rc2_6* check that the signal C is high and RC2 is within its respective thresholds to assign the RC2 rate accordingly. In contrast the original model, demonstrated in Figure 5.16, is composed of several transitions in a closed

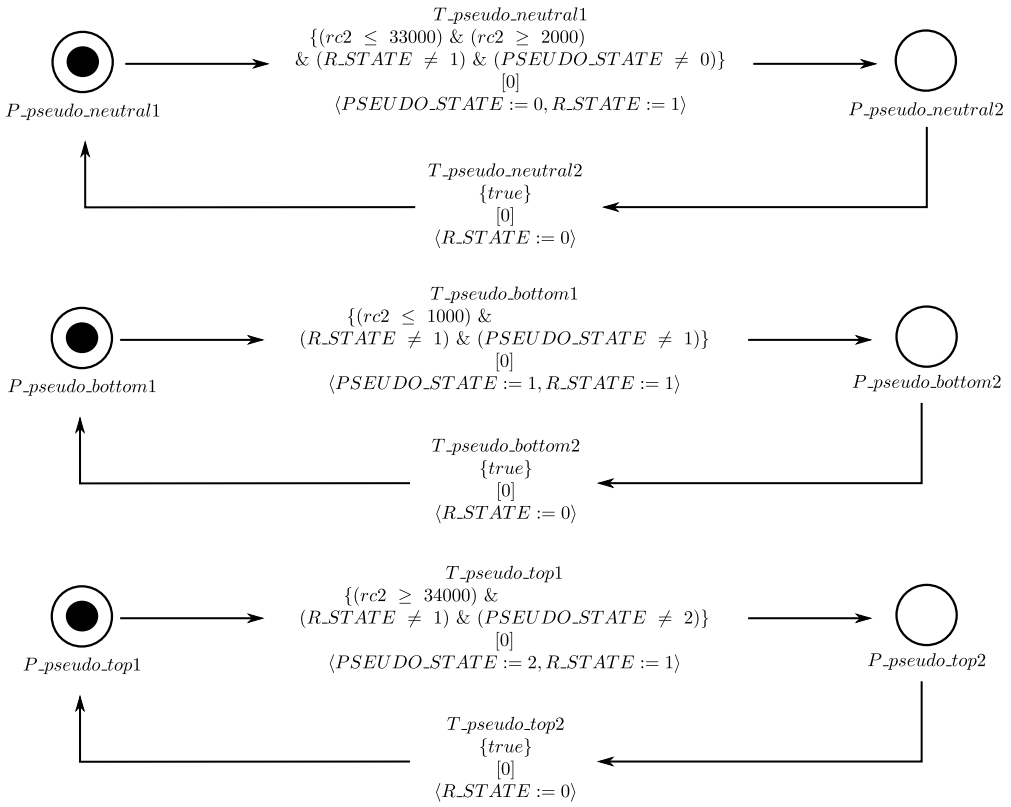


Figure 5.15: Pseudo transitions.

loop form, which iteratively check input conditions and assign the output rate accordingly to approximate the original waveform in a piece-wise linear manner.

Due to the rate variance it is possible for the output voltage to reach values not originally observed in the simulation trace. For example, for the provided input voltage the capacitor voltage level is valid in the range between 0 and 5V. To prevent the model from experiencing the undesired behavior a number of pseudo transitions, as described in [43], were added manually to the model. These transitions, shown in Figure 5.15, set the rate to 0, if the voltage rises too high or drops too low. More information about pseudo transitions and how the process of their generation can be automated is available in the next chapter.

Both models were compared against each other and the results of the simulation for the C-element example are presented in Figure 5.17. The models produce similar waveforms for the data set, that was used to generate them. While the original model yields adequate results on the provided data set, it is not general enough and has limited potential for reusability.

If the control module is switched to an OR element, presented earlier, the original model starts to produce inadequate results. As demonstrated in Figure 5.18, the original model fails to change the rate of an RC circuit appropriately when the signal C switches, which effectively means that the model continues to charge or discharge the capacitor when it should be discharg-

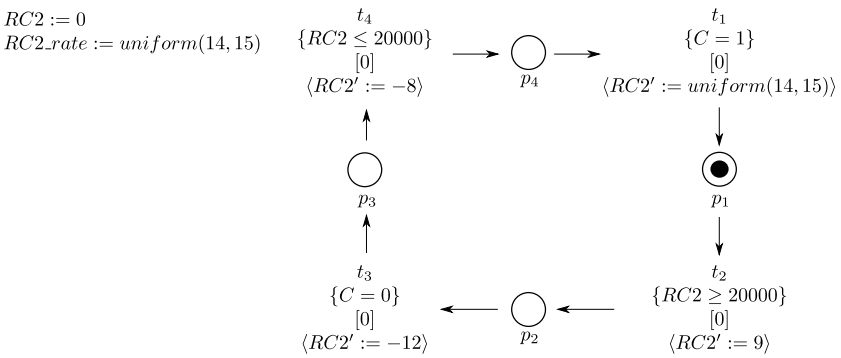
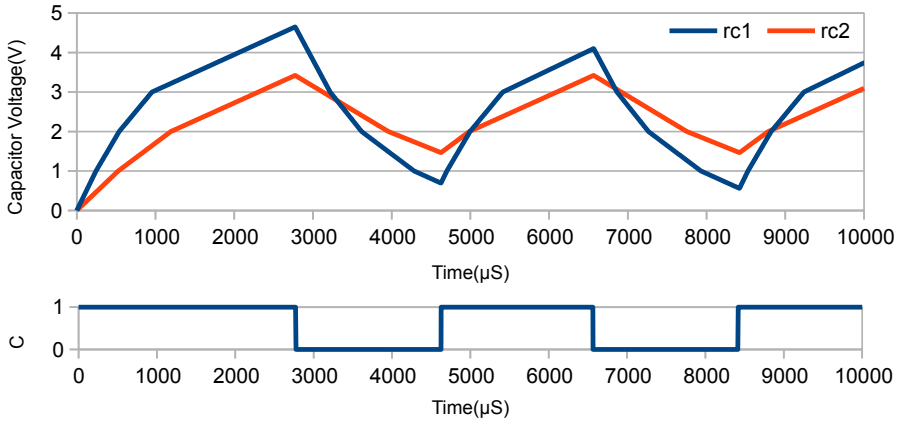
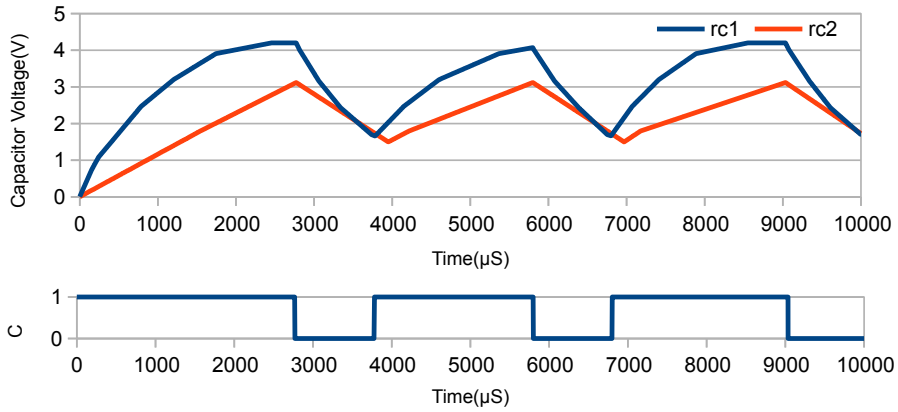


Figure 5.16: Original RC2 model.

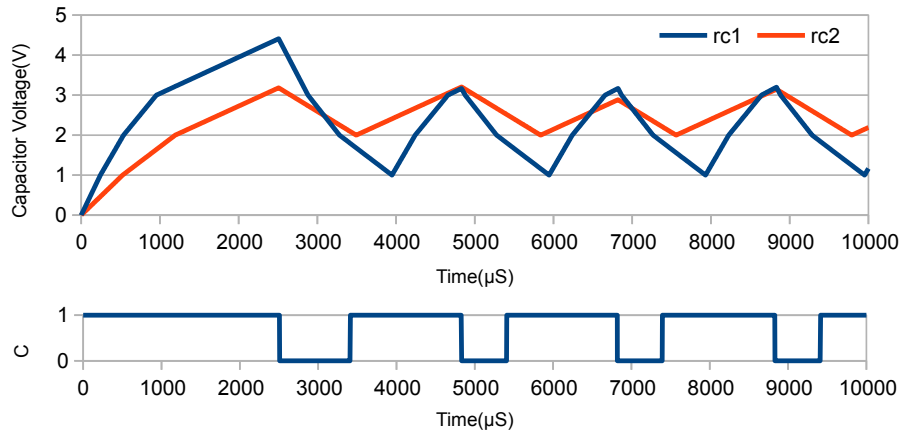


(a) Original model.

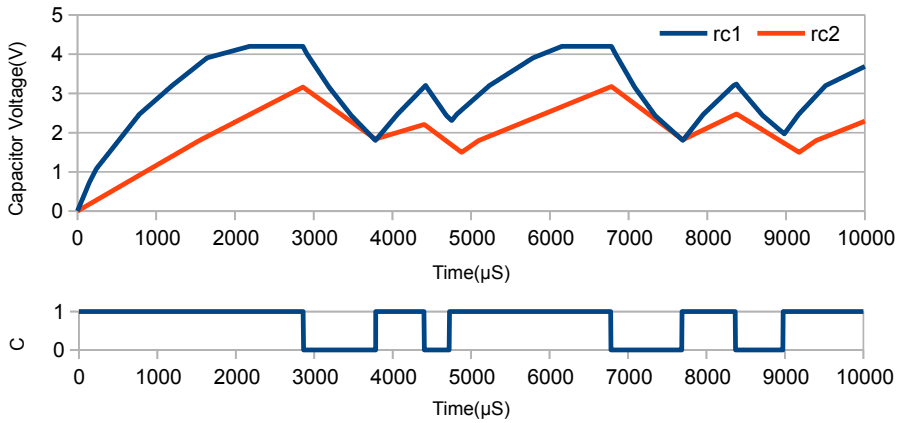


(b) Improved model.

Figure 5.17: C-element simulation results.



(a) Original model.



(b) Improved model.

Figure 5.18: Or-element simulation results.

ing or charging instead. The improved model does not show this issue and produces a more realistic waveform.

5.2.3 *C-element*

While it is possible to convert the STG specification directly to the LPN format to use for whole system simulation or verification the new model generator is also capable of synthesizing control model from a simulation trace. The original data set was used to form a model of the C-element, as illustrated in Figure 5.19.

5.3 Memristor

The memristor is a two-terminal passive element, that combines the behavior of a memory and a resistor [51]. The resistance of the device depends on the magnitude, direction and duration of the voltage applied across its terminals. If the voltage is turned off the memris-

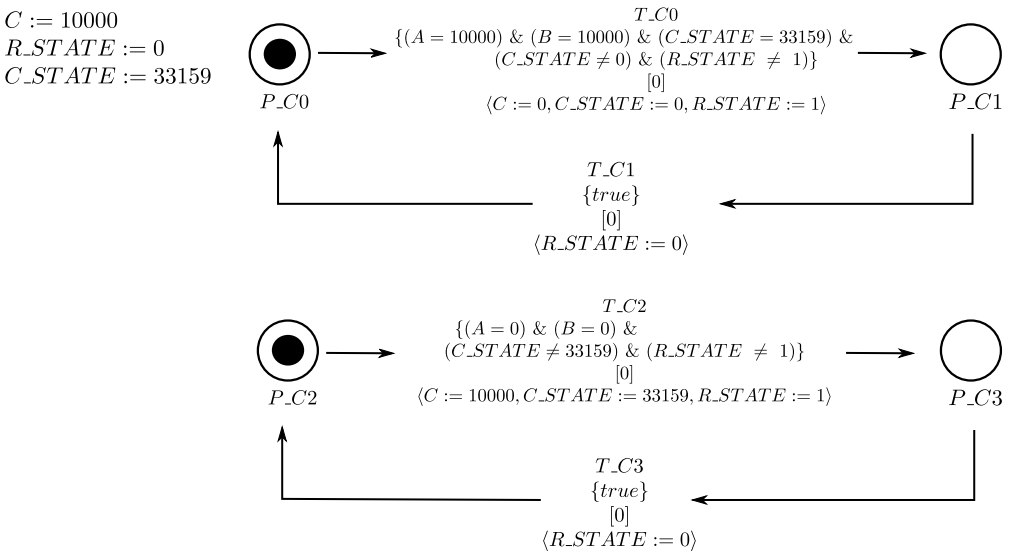


Figure 5.19: C-element LPN model.

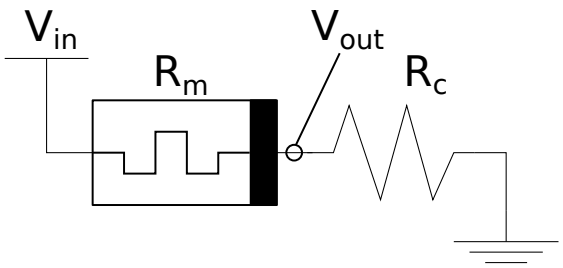


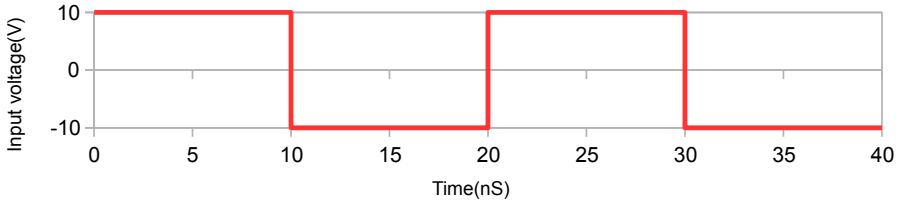
Figure 5.20: Memristor circuit.

tor preserves the resistance value, which allows it to operate as a data storage. Furthermore, not only a memristor has a high endurance of 10^{12} cycles [14], but also multi-value data can be stored in a single device, leading to power and area improvements over the CMOS memory [15].

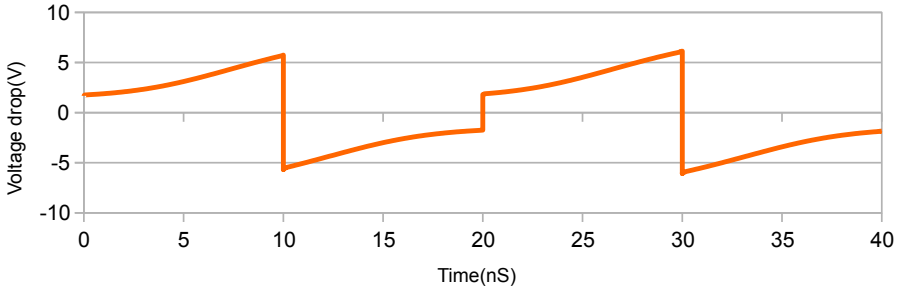
An equivalent memristor tuning circuit [15] consists of an input voltage source, a memristor and a resistor, connected in series, as shown in Figure 5.20. The memristor R_m and the resistor R_c form a voltage divider, which allows to measure the resistance of the memristor as a voltage drop over its terminals. The value of the voltage drop $V_d = V_{out} - V_{in}$ is used by the control circuit, which alternates the input voltage between positive and negative value to adjust the resistance of the memristor.

The simulation waveforms, presented in Figure 5.21, are used to create the model of a memristor. In a manner, similar to the previous example, the model is partitioned between the three modules:

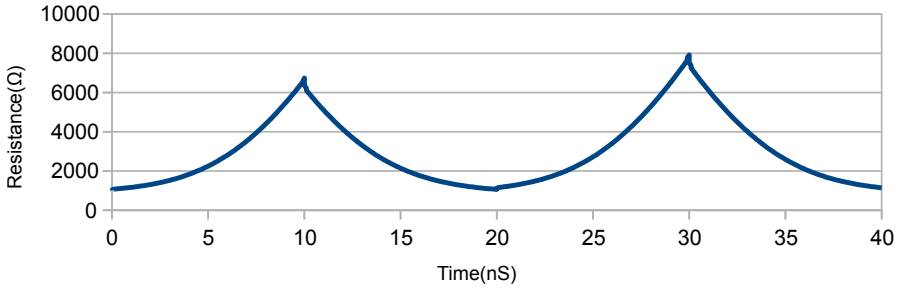
- Control module, which samples the voltage drop and assigns the input voltage accordingly.



(a) Input voltage.



(b) Voltage drop.



(c) Memristor resistance.

Figure 5.21: Memristor waveforms.

- Memristor's resistance, which generates the resistance curve based on the input voltage and resistance.

- Voltage drop, which produces the voltage drop based on the resistance of the memristor.

5.3.1 *Control*

As the main focus of this example is directed towards the memristor model the control LPN was generated from the simulation trace and does not represent the actual control circuit, created by Bunnam et al. The control model, as shown in Figure 5.22, consists of two separate LPNs. Effectively, these LPNs resemble an A2D converter model as they wait for the voltage drop to reach threshold values of *6000* and -1800 mV to assign the input voltage -10 or 10 V respectively.

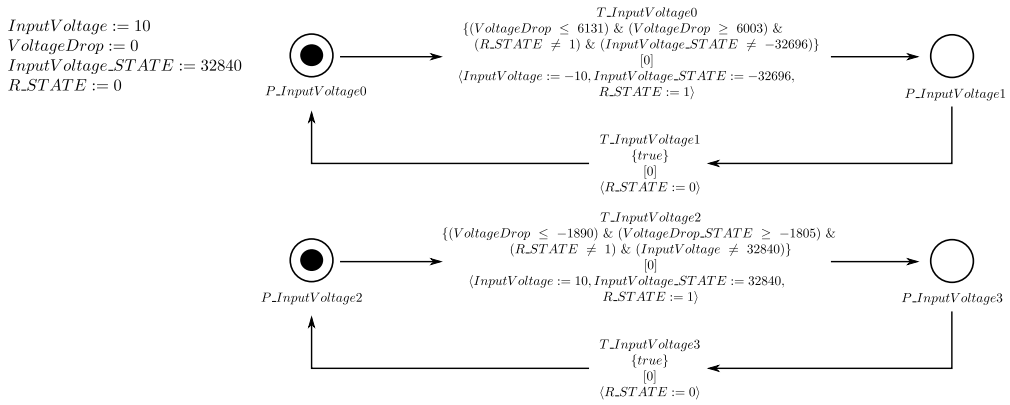


Figure 5.22: Memristor control model.

5.3.2 *Resistance*

The resistance model is constructed similarly to the RC model. The input voltage is selected as the input while the resistance is selected as both the input and the output. To reduce the model size the coefficient of variation in derivative clusterization method is set to 100%, which results in 3 unique states being created per increasing and decreasing areas of the waveform. Additionally, the upper threshold bound in transitions *T_Resistance10* and *T_Resistance4* and the lower threshold bound in transitions *T_Resistance6* and *T_Resistance2* were removed to ensure the correct model behavior. Due to the rate variation it is possible for the model to reach values not originally present in the waveform, which may lead to a deadlock. While it is possible to circumvent this problem by adding the pseudo transitions the main intention of this example is to shown how a potentially unbounded model can be controlled via an analog feedback loop. The resulting model is presented in Figure 5.23.

5.3.3 *Voltage drop*

The voltage drop signal is an unusual signal as it behaves as both an analog and a digital signal. While the input voltage is constant the signal is steadily increasing as a reaction to the memristor's resistance change. However, as the input voltage changes from one value into another the signal also experiences a rapid shift in its value. As a result the derivative based discretization method produces a short duration state with near infinite rate. Normally this state would be filtered out but in this case this state contains an important information about the signal behavior. Consequently, directly applying the model generator is not feasible. Nevertheless two possible solutions to this problem are provided below.

Sample model

The direct method of creating an LPN model for an analog device is by explicitly specifying

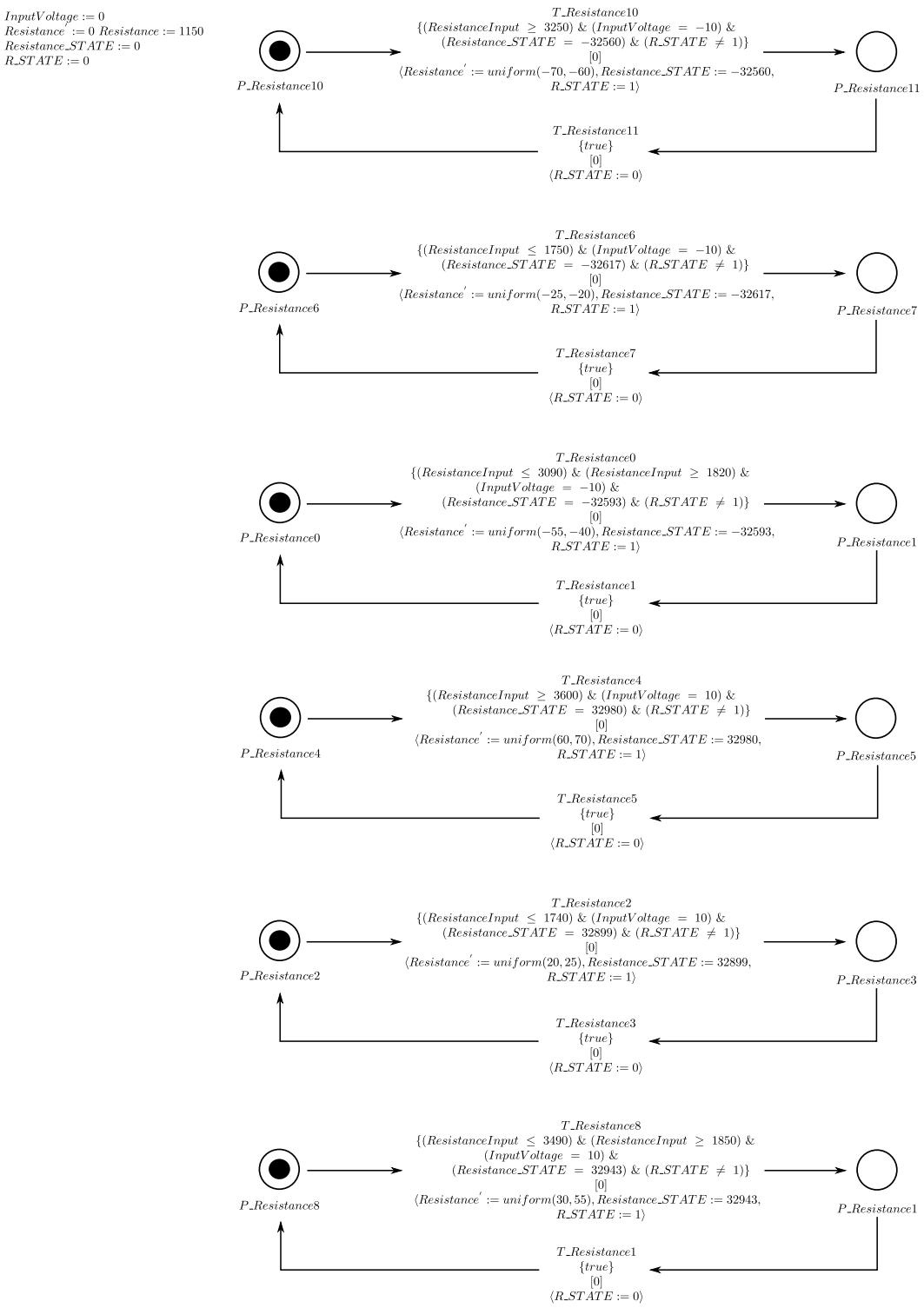


Figure 5.23: Memristor resistance model.

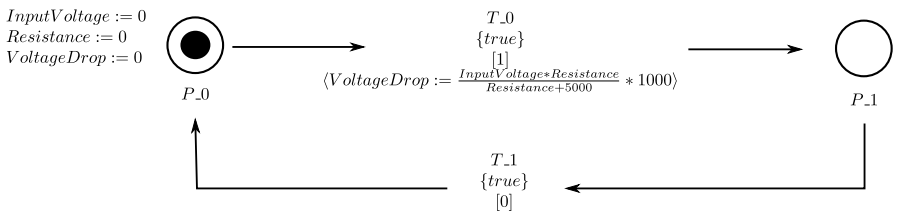


Figure 5.24: Memristor voltage drop sample model.

the transfer function of the component, as shown in Figure 5.24. The assignment condition of an LPN allows to assign a value to a variable as a function of input variables and constants. Every time unit the transition t1 calculates the voltage drop according to the formula $V_d = \frac{V_{in}*R_m}{R_m+R_c}$. The value is also multiplied by a thousand for compatibility with the other models. While this model can produce accurate results it is best used only in simulation as it can lead to a state space explosion.

Hybrid FSM model

To alleviate the problem with state explosion a hybrid approach is used. The main idea of this approach is to use the model generator to generate separate models for continuous parts of the waveform and then bridge them together via an LPN that mimics the discrete behavior of the signal. To create a model of the voltage drop signal the original waveform is partitioned into two pieces: one, when input voltage is high and one, when it is low. In a general case two LPN models will have to be created for these parts, however it is possible to exploit the symmetry of the waveform and create an LPN model only for one part.

The resulting model is shown in Figure 5.25. Transitions *T_VoltageDrop0*, *T_VoltageDrop2* and *T_VoltageDrop4* assign the rate of the signal according to the input resistance. The discrete behavior of the system is produced by the transitions *t_shift_pos* and *t_shift_neg*, which assign the voltage drop to the negative of itself only when a change in the input voltage occurs. Additionally, after the voltage shift occurs the rate of the voltage drop signal is set to 0. This is an important operation as it it ensures that the model operates within the threshold boundaries set for the resistance.

Due to the resistance rate variance it is possible for the resistance to be either above (7920) or below (1280) the observed threshold. Effectively, this means that the operation of the model is undefined in these areas. The simulation trace for the generated model is given in Figure 5.26.

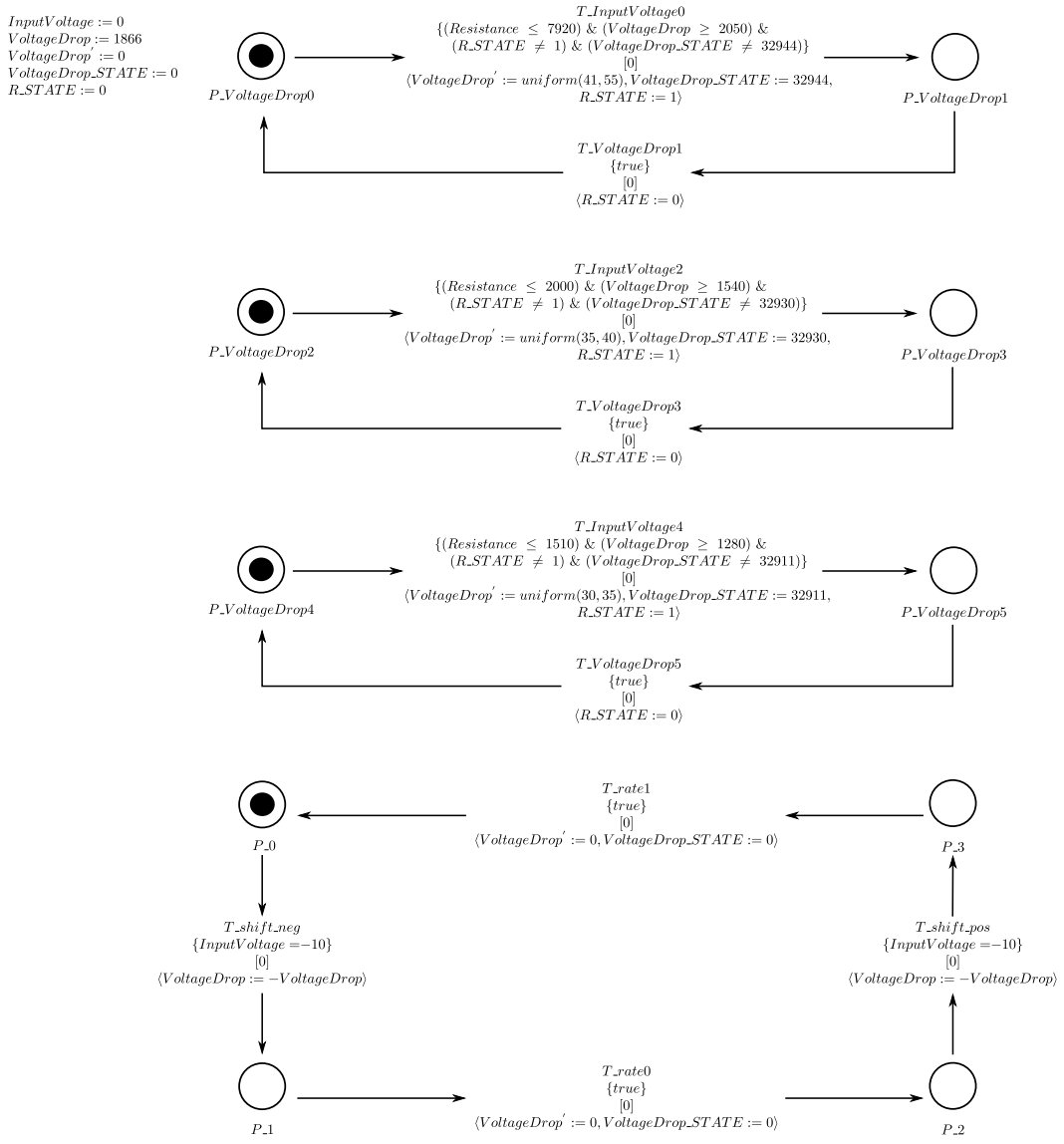
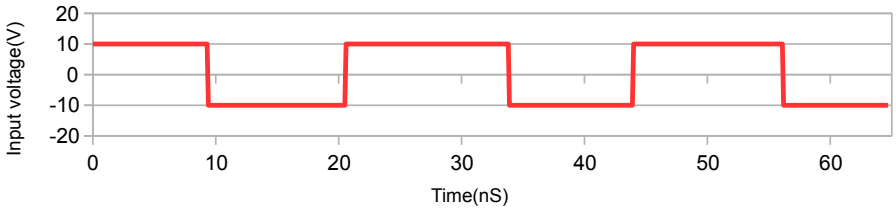
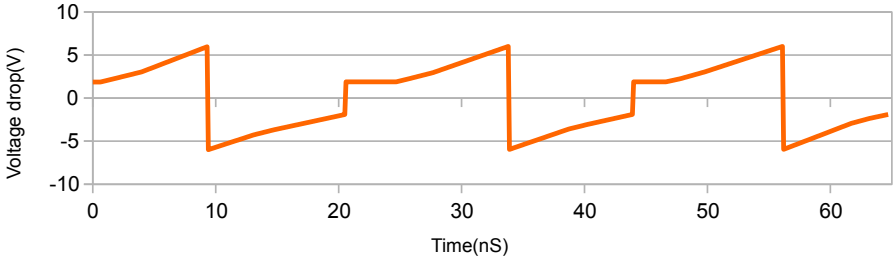


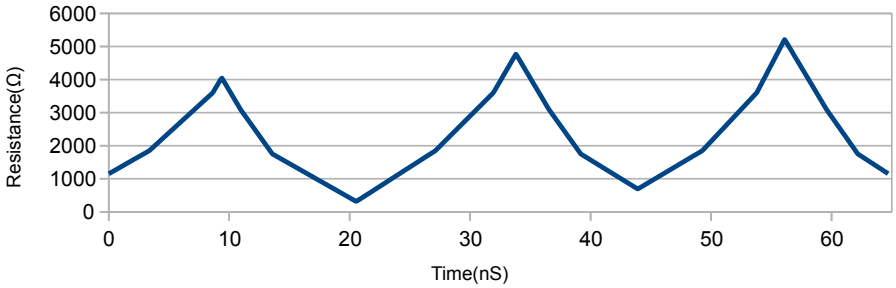
Figure 5.25: Memristor voltage drop state model.



(a) Input voltage.



(b) Voltage drop.



(c) Memristor resistance.

Figure 5.26: Memristor simulation results.

Chapter 6. Conclusions

While neither asynchronous circuits nor formal verification of mixed-signal systems are brand new, the intertwining of these research areas opens new opportunities for AMS designers. The reliance on well-established and developed tools is appealing for the adoption of the described flow by industrial companies. Furthermore, automated model generation is a relatively new research field [33] and has multiple areas of application. The novel concepts and ideas introduced in the model generation framework can be used not only in electrical engineering, but also in cyber-physical systems, biochemical processes, and manufacturing.

6.1 Summary

This dissertation describes a new AMS design flow combining asynchronous circuit design methods, supported by the tool WORKCRAFT, and AMS formal verification, supported by the tool LEMA. As demonstrated in Chapter 2, asynchronous circuits have a clear advantages over the traditional synchronous ones. The automated and formal design approach makes the application of this methodology feasible in an industrial environment. The AMS formal verification, described in Chapter 3, serves as an extension to the existing simulation based verification methods via the means of model generation. The proposed design flow not only combines the existing tools, enhancing the design process of AMS systems, but also extends it by providing possibilities to automatically identify optimization opportunities for the digital control circuit. Additionally, the model generation, based on the LPN formalism, can help in speeding up the design process by reusing models, obtained from the previous system, as well as establish a common design specification framework, suitable for analog and digital engineers. The new model generation approach, explained in Chapter 4, improves the existing method by utilizing data clusterization and process mining techniques. The models, produced by the new model generator, are capable of producing correct behavior, when used with a different control module, as shown in Chapter 5 with the C-element example.

6.2 Future work

A number of additional improvements to the developed model generation framework are considered. The following sections detail usage of additional clustering methods, evaluation of a model fitness, and enhancements to the rule mining algorithm.

6.2.1 *Additional clusterization methods*

The new model generation framework relies heavily on clusterization for discretization and filtering of data. While the currently employed agglomerative clustering method is effective, the cubic time complexity of the algorithm makes it slow and inefficient for even medium data sets. A number of additional clustering methods can be used to alleviate this problem.

6.2.1.1 K-Means clustering

K-means clustering [48] is a well known data clustering algorithm. The algorithm requires the user to explicitly specify the number of clusters to use and initializes these clusters with randomly chosen center points from the data set. After that all the data points are distributed between the closest clusters, according to a metric function, and the clusters centers are recalculated. The algorithm proceeds to iterate between distributing the data points and adjusting clusters' centers until their values reach a steady value.

This clustering method features linear time complexity when using Lloyd's algorithm [47]. While in general case predicting the number of clusters requires good understanding of the data set, detecting the number of clusters for certain signals, such as DMV signals, is rather straightforward. Furthermore, this method can useful to limit the number of unique output states produces and thus provide explicit means of control over model complexity.

6.2.1.2 DBScan

A good alternative to the K-means clustering algorithm is DBScan [26]. DBScan begins with an arbitrary starting data point that has not been visited. If there is a sufficient number of data points, specified by the *minPts* parameter, within the ε neighborhood of this point, the current data point becomes the first point in the new cluster. Otherwise, the point is labeled as noise, but can still become the part of the cluster. In both cases the point is marked as visited. The first point in the cluster is used to add additional data points, residing within ε distance. This process is repeated for all newly added data points to the cluster group. If no more points

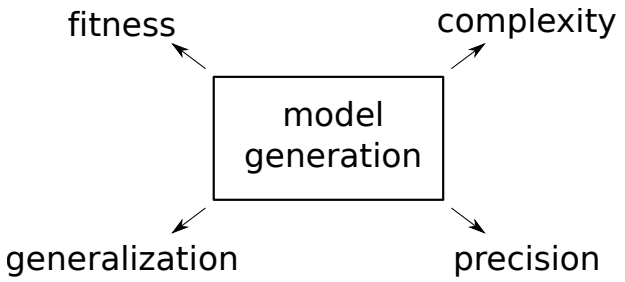


Figure 6.1: Quality dimensions for model generation.

can be added the algorithm selects a new unvisited data point and repeats the process of creating a cluster group.

While this method does not require the designer to explicitly specify the number of cluster groups, the algorithm requires to define parameters of cluster such as the minimum number of points and distance between the data points. As a result this algorithm cannot be used on a broad set of examples using the same parameters and therefore is less flexible than agglomerative clustering.

6.2.1.3 Sliding window and bottom-up

The sliding window and bottom-up (SWAB) [34] algorithm algorithm keeps a small buffer, sufficient to create 5-6 data segments, and applies the Bottom-Up algorithm to group the data. The data, corresponding to the leftmost segment, is removed from the buffer and more data points are read in. The number of the data points read in depends on the structure of the data and is determined by a sliding window function. This process repeats until no data is available.

Effectively, the SWAB algorithm creates a semi-global view of the data using a sliding window technique and refines the segmented data via the Bottom-Up algorithm. The algorithm can be seen as operating on a continuum between the two extremes of Sliding Windows and Bottom-Up. Although the algorithm does not help to minimize the computational complexity it requires only a small constant amount of memory.

6.2.2 *Evaluating generated models*

The goal of process mining and, consequently, model generation is producing process models by considering only operational records [13]. While the traditional process mining deals with discrete event logs, discretization of analog data adds another degree of complexity and variation to the model generation process. In return this affects the measurements for evaluating the obtained models.

As shown in Figure 6.1 there are four quality dimensions used to describe the results of discovery techniques:

- **Fitness.** Replay fitness characterizes the ability of the model to reproduce the behavior of the original simulation trace. There are two possible methods to measure the fitness of the obtained model. The first method relies on calculating the cross-correlation between the original output signal and the output signal produced by the model given the same input stimulus. However, when generating the output signal extra care for selecting rates and delays on transitions is needed as those can effect the resulting waveform. The second method operates on untimed sequences of states. As the original waveform is discretized it is transformed into a sequence of states, or, essentially, an event log, which is used to create a model. As a result it is possible to analyze the state graph, produced by the model, and extract the sequence of the output states to compare it against the original sequence.

- **Complexity.** The complexity dimension effectively describes the model's structure and size and is directly related to how good a model can be perceived by a human. While there is no uniform metric for estimating a complexity of a model, it can be calculated as the total number of nodes, such as places and transitions, and interconnections between them.

- **Precision.** Precision quantifies the fraction of the behavior of the model not seen in the original waveform. For example, a simple model, as shown in Figure 6.2, can produce the original waveform for the variable **V**. However, this model can also produce any arbitrary waveform without any restrictions, which makes this model not very useful. Existing methodologies for estimating a model precision, described in [13, 6], are best suited for models with a lot of concurrency. Models, produced by the new model generator, generally have only one transition enabled at a time due to guard conditions, hence application of these methodologies is not optimal. One possible solution, at least for analog signals, is to measure precision as the relation between the original signal and the generated signal ranges. Furthermore, since precision metric is essentially an indication of how much a model can deviate from an acceptable behavior it is possible to impose a hard limit on this deviation via model properties.

- **Generalization.** Finally, the generalization metric assesses the extent to which a model can reproduce future behavior of a system. In the case of AMS systems this means that

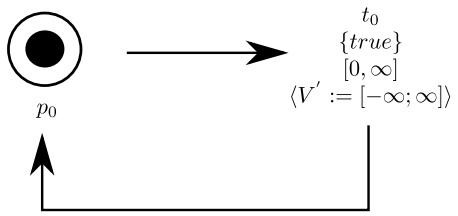


Figure 6.2: Imprecise model.

models of an analog environment can be reused with a different control module as was demonstrated in the C-element example. There is no reliable universal method for measuring this metric. However, a similar approach, employed in machine-learning, can be used. A large enough set of simulation traces can be split in two parts: one, that is used to generate the model, and another, used to test the model.

Estimating a model fitness is a difficult, yet crucial problem. Not only it is necessary to have a method of comparing models, produced by the new model generator, against the models of the existing one, but is also important to evaluate the resulting models to guide the model generation process. While there are multiple dimensions to the model optimization, the most critical components of a metric function are:

- Fitness. The model must produce the behavior of the original system.
- Precision. The model must stay within specified bounds.
- Complexity. The model must be compact and human readable.

6.2.3 *Improving rule mining*

The new model generator features a number of complex methods containing dozens of internal parameters. These parameters affect not only the operation of any particular algorithm but also determine the order and applicability of algorithms. For example, the order in which conflict resolution methods are used in the rule mining stage is determined by a metric function of each method. Currently the metric cost is defined explicitly by a designer, which requires some intuition and understanding of how LPN models are synthesized. However, the ultimate goal of this work is to provide automated means for model generation, which means that the new framework has to determine "knobs" for optimal operation by itself.

Evaluation of a model fitness, discussed in the previous section, can provide the necessary feedback and help in improving the rule mining process. Specifically, the improved rule mining algorithm can construct a number of models by applying conflict resolution methods in different

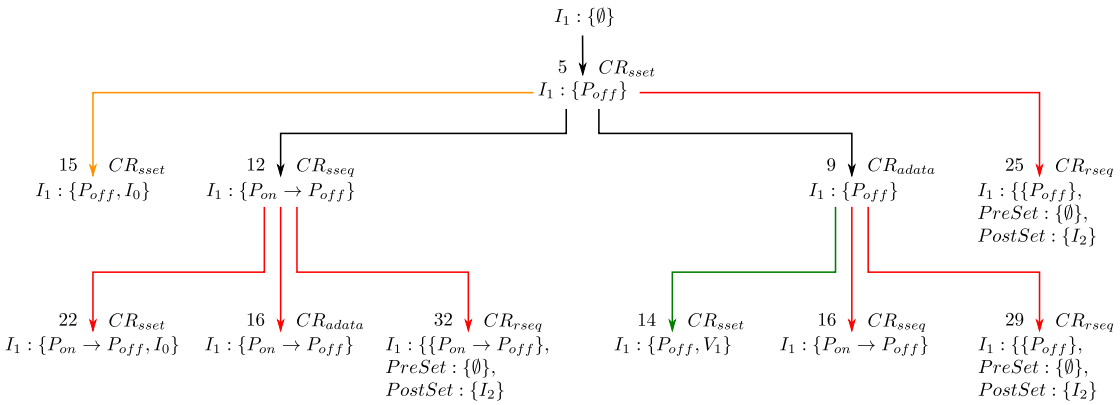


Figure 6.3: Branch and bound tree graph.

order and then chose the model with the best fitness score. Effectively, this means that the rule mining module has to perform state space search to determine the ideal combination of parameters values. A couple of methods for efficient state space search are provided below. It is important to note that model score depends only on the model complexity as both fitness and precision metrics can be approximated as binary values, indicating that the resulting model is operational.

6.2.3.1 Branch and bound

Branch and bound [40] is an algorithm design paradigm for optimization problems. The goal of the algorithm is to find a candidate solution that minimizes or maximizes the value of a real-valued function by means of state space search. The state space is organized as a rooted tree. A branch and bound algorithm explores branches of this tree, which represent a subset of the complete solution, and discards those that exceed estimated bounds on the optimal solution.

For resolving pattern conflicts the rule mining stage employs greedy algorithm, which picks the resolution method with the lowest metric cost. This can result in a suboptimal model being generated. Figure 6.3 illustrates how branch and bound algorithm can be used to address this problem. The tree graph for the rule pattern I_1 of the inductor current model, discussed in Chapter 4, represents transformations of the rule pattern after application of different conflict resolution methods:

- Resolve by state set (CR_{sset}) has a cost of 5 for adding an input state and cost 10 for adding an output state to the patter,
- Resolve by set sequence (CR_{sseq}) has a cost of 7 per sequence added,
- Resolve by analog data (CR_{adata}) has a cost of 4 for adding new states to the data pattern,

- Resolve by rule sequence (CR_{rseq}) has a cost of 20.

Initially, an upper bound on the potential solution is found. This is achieved by applying the CR_{sset} twice, which results in the rule pattern $I_1 : \left\{P_{off}, I_0\right\}$[1], marked by an orange line, with a score of 15. This value is used to make branching decisions and prevent the rule mining algorithm from exploring solutions with worse score. As a result application of the CR_{rseq} method to the RP $I_1 : \left\{P_{off}\right\}$, marked by a red line, does not take place as the obtained RP has higher score than the established limit. However, it is possible to use methods CR_{sseq} and CR_{adata} as neither exceeds the upper bound. Exploring the branch for the RP $I_1 : \left\{P_{on} \rightarrow P_{off}\right\}$ reports no improvements over the solution found so far. The branch for the RP $I_1 : \left\{P_{off}\right\}$ after application of the method CR_{adata} [2] contains an optimal solution $I_1 : \left\{P_{off}, V_1\right\}$,marked by green line.

6.2.3.2 Genetic algorithm

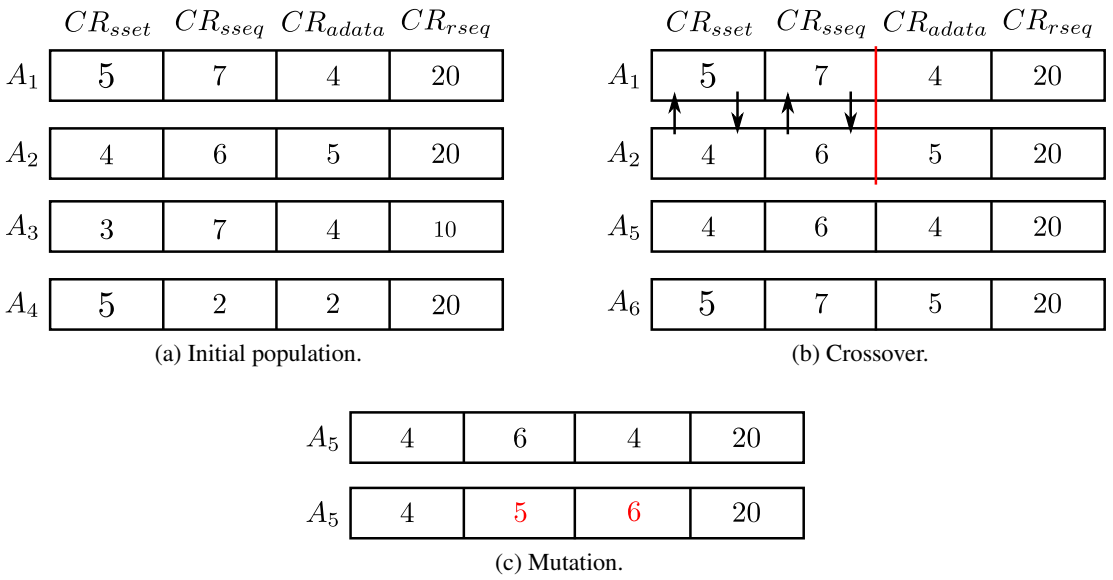
A genetic algorithm [50] is a heuristic search algorithm inspired by the evolutionary processes found in nature. The main idea of a genetic algorithm is to select the fittest individuals from a population and produce offspring, which inherit characteristics of their parents and are used in the next generation. There are five main phases of a genetic algorithm:

- **Initial population.** The first phase is the creation of a set of individual solutions to the problem, called population. An individual is characterized by a set of parameters, called genes. Effectively, various parameters of the rule mining algorithm, such as the metric cost of conflict resolution methods, can be formulated as a genetic code of a single individual.

- **Fitness function.** The fitness function evaluates each individual in a population and assigns a fitness score it. The probability that an individual will be selected for reproduction depends on this metric. Calculating a fitness score for a set of rule patterns can be done by combining the costs of all used conflict resolution algorithms, used to create these rule patterns. Alternatively, it is possible to use a more sophisticated scoring approach based on calculation of the fitness of the resulting model.

- **Selection.** After the evaluation of individuals it is necessary to select the fittest individuals and let them pass their genes to the next generation.

[1] This RP has no conflicts with other RPs or DPs and can be used for model generation.
[2] This method does not change the RP as it only adds new states to its DP.



(a) Initial population.

(b) Crossover.

(c) Mutation.

Figure 6.4: Genetic algorithm for rule mining.

- **Crossover.** Every pair of the fittest individuals exchanges genes to create offspring. The crossover point determines the number of genes that can be exchanged.

- **Mutation.** Finally, the genes of certain new offspring can be subject to a mutation with a low random probability. This effectively means that parameters, representing the mutated genes, are assigned a new random value.

The described algorithm can be used to improve the rule mining stage and automatically find the optimal parameter values, as described in Figure 6.4. The initial population is formed by creating a set of individual solutions. Every individual is represented by its genetic code, comprising of a vector of metric costs of conflict resolution methods. These metric costs are selected arbitrarily or using a heuristic. After that every individual is used to generate an LPN model, which is evaluated via a fitness function, and several individuals with highest fitness score a selected to produce offspring. This is a two step process. Every pair of "parents" exchange genes until a selected crossover point, as illustrated in Figure 6.4b. For example, in this case the most fit individuals A_1 and A_2 are produce children A_5 and A_6 by swapping parameters for methods CR_{sset} and CR_{sseq}. Afterwards, a random mutation happens in children, as shown in Figure 6.4c, which causes parameters CR_{sseq} and CR_{adata} to be assigned new random values. Finally, the offspring are added to the population to be used for model generation. This process repeats until the population has converged, which means that offspring do not produce significantly different results from their previous generation.

6.2.4 *Pseudo-transitions*

Due to the piece-wise linear approximation it is possible for the analog signals, produced by an LPN model, to reach values not present in the original simulation trace. In certain cases this can be an undesired behavior as it can violate physical principles of the system. For example, the voltage on a capacitor in the C-element example cannot exceed the supply voltage. This effectively means that this analog signal is only allowed to operate within specific boundaries. To cope with this problem a similar approach to pseudo-regions [43] can be used.

At the first step the operational boundaries of an analog signal have to be found. This can be achieved by calculating minimum and maximum values, located in a simulation waveform. After that a new variable, *PSEUDO_STATE*, is added to the generated LPN model. This variable represents in which region the analog is located and takes the following values:

- 0 means that the signal is in between specified low and upper boundaries,
- 1 means that the signal is below than the low boundary,
- 2 means that the signal is higher than the upper boundary.

This variable is added as an additional check to a guard condition of all transitions in the model with the exception of reset transitions. For the transitions that lead to the assignment of positive rates this variable is checked against the value of 2, while for negative rates this variable is checked against the value of 1. Additionally, three LPNs are added to the model that set this variable to the appropriate value and also set the rate of the signal to 0, if it exceeds one of the boundaries. This effectively means that if the signal reaches upper boundary its rate is set to 0 and only transitions that can set rate to a negative value can be enabled. The illustration of this approach is given in Figure 5.14 in Chapter 5

6.2.5 *Annotating STG with timing information*

The final improvement to the described methodology is the annotation of an STG specification with the timing information. The goal of the AMS design flow, presented in Chapter 3, is to test the control module, formulated as an STG, under an analog environment. Although, an STG can be automatically converted into an LPN to be used for formal verification, the resulting model is missing critical delay information that can affect the operation of the control module. This information can be extracted from a circuit implementation and used to infuse the LPN model with additional delays on transitions.

Bibliography

[1] Modeling coilcraft rf inductors. http://www.ing.unp.edu.ar/electronica/asignaturas/ee016/anexo/l-coilcraft-pspice.pdf.

[2] ATACS – Automated Timed Asynchronous Circuit Synthesis. http://www.async.ece.utah.edu/ATACS/.

[3] LEMA homepage. http://www.async.ece.utah.edu/LEMA/.

[4] PETRIFY homepage. http://www.cs.upc.edu/~jordicf/petrify/.

[5] WORKCRAFT homepage. http://workcraft.org/.

[6] Arya Adriansyah, Jorge Munoz-Gama, Josep Carmona, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Alignment based precision checking. In Marcello La Rosa and Pnina Soffer, editors, *Business Process Management Workshops*, pages 137–149, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] M. Althoff, A. Rajhans, B. H. Krogh, S. Yaldiz, X. Li, and L. Pileggi. Formal verification of phase-locked loops using reachability analysis and continuization. *Commun. ACM*, 56(10):97–104, 2013.

[8] J. Audi. Navigating the path to a successful ic switching regulator design. Tutorial at Int. Solid-State Circuits Conference (ISSCC).

[9] D. Baba. Benefits of a multiphase buck converter. *Analog Applications Jorunal, High Performance Analog Products, Texas Instruments*, pages 1–8, 2012.

[10] S. Batchu. Automatic extraction of behavioral models from simulations of analog/mixed-signal (AMS) circuits. Master's thesis, University of Utah, 2010.

[11] J. Beaumont, A. Mokhov, D. Sokolov, and A. Yakovlev. High-level asynchronous concepts at the interface between analog and digital worlds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):61–74, Jan 2018.

[12] W. Belluomini, C. J. Myers, and H. P. Hofstee. Verification of delayed-reset domino circuits using ATACS. In *Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 3–12, 1999.

[13] Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In Robert Meersman, Hervé Panetto, Tharam Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and Isabel F. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM 2012*, pages 305–322, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] T. Bunnam, A. Soltan, D. Sokolov, and A. Yakovlev. Pulse controlled memristor-based delay element. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sept 2017.

[15] T. Bunnam, A. Soltan, D. Sokolov, and A. Yakovlev. An excitation time model for general-purpose memristance tuning circuit. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018.

[16] T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[17] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on information and Systems*, 80(3):315–325, 1997.

[18] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavango, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Springer-Verlag Berlin Heidelberg, 2002.

[19] A. d. Gennaro, P. Stankaitis, and A. Mokhov. A heuristic algorithm for deriving compact models of processor instruction sets. In *2015 15th International Conference on Application of Concurrency to System Design*, pages 100–109, June 2015.

[20] René David and Hassane Alla. *Timed Hybrid Petri Nets*, chapter 6, pages 231–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[21] W. Denman, B. Akbarpour, S. Tahar, M. H. Zaki, and L. C. Paulson. Formal verification of analog designs using MetiTarski. In *Proc. Formal Methods in Computer-Aided Design*, pages 93–100, 2009.

[22] V. Dubikhin, C. Myers, D. Sokolov, I. Syranidis, and A. Yakovlev. Advances in formal methods for the design of analog/mixed-signal systems. In *Proc. Design Automation Conference (DAC)*, 2017.

[23] V. Dubikhin, D. Sokolov, C. J. Myers, A. Mokhov, and A. Yakovlev. Model discovery for analog/mixed-signal circuits. In *FAC 2017; Frontiers in Analog CAD*, pages 1–6, July 2017.

[24] V. Dubikhin, D. Sokolov, A. Yakovlev, and C. J. Myers. Design of mixed-signal systems with asynchronous control. *IEEE Design & Test*, 33(5):44–55, 2016.

[25] D. A. Edwards and W. B. Toms. Design, automation and test for asynchronous circuits and systems. Technical report, Information Society Technologies, 2004.

[26] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[27] A. Fisher. *Efficient, sound formal verification for analog/mixed-signal circuits*. PhD thesis, University of Utah, 2015.

[28] A. N. Fisher, S. Batchu, K. Jones, D. Kulkarni, S. Little, D. Walter, and C. J. Myers. LEMA: A tool for the formal verification of digitally-intensive analog/mixed-signal circuits. In *Proc. International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1017–1020, 2014.

[29] G. Frehse, B. H. Krogh, and R. A. Rutenbar. Verifying analog oscillator circuits using forward/backward abstraction refinement. In *Proc. Design, Automation & Test in Europe (DATE)*, pages 257–262, 2006.

[30] G. Frehse, C. Le Guernic, A. Donze, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SPACEEX: Scalable verification of hybrid systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.

[31] D. Grabowski, C. Grimm, and E. Barke. Semi-symbolic modeling and simulation of circuits and systems. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, pages 983–986, 2006.

[32] M. R. Greenstreet and S. Yang. Verifying start-up conditions for a ring oscillator. In *Proc. ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 201–206, 2008.

[33] Y Huang, A Verbraeck, and M Seck. Graph transformation based simulation model generation. *Journal of Simulation*, 10(4):283–309, 2016.

[34] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 289–296, Nov 2001.

[35] V. Khomenko. A usable reachability analyser. Technical report, Newcastle University, 2009.

[36] D. J. Kinniment. *Synchronization and arbitration in digital systems*. Wiley, 2008.

[37] S. Kowalewski, M. Garavello, H. GuÃ©guen, G. Herberich, R. Langerak, B. Piccoli, J. W. Polderman, and C. Weise. *Hybrid automata*, chapter 3, pages 57–86. Cambridge University Press, 2009.

[38] D. Kulkarni. Improved model generation and property specification for analog/mixed-signal circuits. Master's thesis, University of Utah, 2013.

[39] Dhanashree Kulkarni, Satish Batchu, and Chris Myers. Improved model generation of ams circuits for formal verification. In *2011 Virtual Worldwide Forum for PhD Researchers in Electronic Design Automation*, 2011.

[40] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14(4):699–719, August 1966.

[41] C. Le Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *LNCS*, pages 540–554. Springer, 2009.

[42] B. C. Lim, J. E. Jang, J. Mao, J. Kim, and M. Horowitz. Digital analog design: Enabling mixed-signal system validation. *IEEE Design & Test*, 32(1):44–52, 2015.

[43] S. Little. *Efficient modeling and verification of analog/mixed-signal circuits using labeled hybrid petri nets*. PhD thesis, University of Utah, 2008.

[44] S. Little, D. Walter, C. J. Myers, R. Thacker, S. Batchu, and T. Yoneda. Verification of analog/mixed-signal circuits using labeled hybrid Petri nets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):617–630, 2011.

[45] Scott Little, David Walter, Kevin Jones, Chris Myers, and Alper Sen. Analog/mixed-signal circuit verification using models generated from simulation traces. *International Journal of Foundations of Computer Science*, 21(02):191–210, 2010.

[46] D. Lloyd and R. Illman. Scan insertion and atpg for c-gate based asynchronous designs. Synopsys User Group (SNUG), 2014.

[47] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.

[48] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[49] I. B. Makhlouf and S. Kowalewski. Networked cooperative platoon of vehicles for testing methods and verification tools. In G. Frehse and M. Althoff, editors, *Proc. Int. Workshop on Applied veRification for Continuous and Hybrid Systems (ARCH)*, volume 34 of *EPiC Ser. in Comp.*, pages 37–42, 2015.

[50] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[51] S. P. Mohanty. Memristor: From basics to deployment. *IEEE Potentials*, 32(3):34–39, May 2013.

[52] A. Mokhov, V. Khomenko, D. Sokolov, and A. Yakovlev. Opportunistic merge element. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 116–123, May 2015.

[53] R. Narayanan, B. Akbarpour, M. Zaki, S. Tahar, and L. Paulso. Formal verification of analog circuits in the presence of noise and process variation. In *Proc. Design, Automation & Test in Europe (DATE)*, pages 1309–1312, 2010.

[54] L. V. Nguyen and T. T. Johnson. Benchmark: DC-to-DC switched-mode power converters (buck converters, boost converters, and buck-boost converters). In G. Frehse and M. Althoff, editors, *Proc. International Workshop on Applied veRification for Continuous and Hybrid Systems (ARCH)*, volume 34 of *EPiC Series in Computing*, pages 19–24, 2015.

[55] S. M. Nowick and M. Singh. Asynchronous design - part 2: Systems and methodologies. *IEEE Design Test*, 32(3):19–28, June 2015.

[56] D. Perry and H. Foster. *Applied formal verification: For digital circuit design*. Electronic Engineering. McGraw-Hill, 2005.

[57] Ed Petrus. Trends in analog/mixed-signal design tools. `http://www.isqed.org/English/Archives/2013/keynotes/Ed_Petrus_Trends_In_Analog_Mixed_Signal_Design_Tools_ISQED2013.pdf`, 2013.

[58] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev. Automated verification of asynchronous circuits using circuit petri nets. In *2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 161–170, April 2008.

[59] Abraham Pressman. *Switching Power Supply Design*. McGraw-Hill, Inc., New York, NY, USA, 2 edition, 1998.

[60] C. Radojicic and C. Grimm. Formal verification of mixed-signal designs using extended affine arithmetic. In *Proc. PhD Research in Microelectronics and Electronics (PRIME)*, 2016.

[61] Lior Rokach and Oded Maimon. *Clustering Methods*, pages 321–352. Springer US, Boston, MA, 2005.

[62] D. Sokolov, A. de Gennaro, and A. Mokhov. Reconfigurable asynchronous pipelines: From formal models to silicon. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1562–1567, March 2018.

[63] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev. Benefits of asynchronous control for analog electronics: multiphase buck case study. In *Proc. Design, Automation & Test in Europe (DATE)*, 2017.

[64] D. Sokolov, V. Khomenko, A. Mokhov, A. Yakovlev, and D. Lloyd. Design and verification of speed-independent multiphase buck controller. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 29–36, 2015.

[65] D. Sokolov, A. Mokhov, A. Yakovlev, and D. Lloyd. Towards asynchronous power management. In *Proc. IEEE Faible Tension Faible Consommation(FTFC)*, 2014.

[66] Danil Sokolov, Victor Khomenko, and Andrey Mokhov. Workcraft : Ten years later. 2016.

[67] J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design*. Springer Us, 2013.

[68] S. Steinhorst and L. Hedrich. Improving verification coverage of analog circuit blocks by state space-guided transient simulation. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, pages 645–648, 2010.

[69] M. Tadeusiewicz and S. Halgas. A method for finding multiple DC operating points of short channel CMOS circuits. *Circuits, Systems, and Signal Processing*, 32(5):2457–2468, 2013.

[70] S. K. Tiwary, A. Gupta, J. R. Phillips, C. Pinello, and R. Zlatanovici. First steps towards SAT-based formal analog verification. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2009.

[71] T. Towers. Practical design problems in transistor DC/DC converters and DC/AC inverters. *Proc. IEE*, 1959.

[72] Stephen H. Unger. *Asynchronous Sequential Switching Circuit*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1983.

[73] D. Walter, S. Little, C. J. Myers, N. Seegmiller, and T. Yoneda. Verification of analog/mixed-signal circuits using symbolic methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2223–2235, 2008.

[74] C. Yan and M. R. Greenstreet. Circuit level verification of a high-speed toggle. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, pages 199–206, 2007.

[75] C. Yan and M. R. Greenstreet. Formal verification of an arbiter circuit. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 165–175, 2010.

[76] M. H. Zaki, I. M Mitchell, and M. R. Greenstreet. DC operating point analysis – a formal approach. In *Proc. Frontiers in Analog CAD (FAC)*, 2009.

[77] M. H. Zaki, S. Tahar, and G. Bois. Formal verification of analog and mixed signal designs: A survey. *Microelectron. J.*, 39(12):1395–1404, 2008.

[78] M. Zwolinski and D. A. Crutchley. Using evolutionary and hybrid algorithms for DC operating point analysis of nonlinear circuits. In *Proc. of Congress on Evolutionary Computation (CEC)*, pages 753–758, 2002.