IMPROVED MODEL GENERATION AND PROPERTY SPECIFICATION FOR ANALOG/MIXED-SIGNAL CIRCUITS

by

Dhanashree R. Kulkarni

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

The University of Utah

August 2013

Copyright © Dhanashree R. Kulkarni 2013

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

This thesis of **Dhanashree R. Kulkarni**

has been approved by the following supervisory committee members:

Chris Myers, Chair **06/07/2013**
Date Approved

Kenneth Stevens, Member **06/11/2013**
Date Approved

Scott Little, Member **06/03/2013**
Date Approved

and by **Gianluca Lazzi**, Chair of the Department of **Electrical and Computer Engineering** and by Donna M. White, Interim Dean of The Graduate School.

ABSTRACT

This document describes an improved method of formal verification of complex *analog/mixed-signal* (AMS) circuits. Currently, in our `LEMA` tool, verification properties are encoded using *labeled Petri net* (LPN). These LPNs are generated manually, a tedious process that requires the user to have considerable familiarity with the tool. To eliminate this time-consuming process, our `LEMA` tool is extended to include a translator that converts properties written in a property specification language to LPNs. New methods are also implemented to separate the transient period from the stable output period, thus improving the generated model. Also, the current methodology generates the circuit models for the input values used during the simulation of the circuit. So, models generated for other control input values are not accurate. In this case, accuracy of the generated models is improved by using a linear abstraction method like *interpolation*.

To my advisor, Chris

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **vii**

ACKNOWLEDGEMENTS . . . . **viii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Model Generation 3
1.2 Property Generation 5
1.3 Contributions 5
1.4 Thesis Overview 6

2. BACKGROUND **8**

2.1 LEMA 8
2.2 Labeled Petri Net (LPN) 10
2.2.1 LPN Syntax 11
2.2.2 LPN Semantics 12
2.3 LPN Generation Process 13
2.4 Verification Property 17

3. PROPERTY LANGUAGE TRANSLATOR **19**

3.1 Motivation 19
3.2 Real-Time SVA 20
3.3 Property Language 23
3.4 Property Compiler 28
3.5 Examples 29
3.6 Verification 35

4. MODEL GENERATION AND INTERPOLATION **40**

4.1 Motivation 40
4.2 Modeling Transient Behavior 44
4.3 Interpolation 49

5. CONCLUSIONS **53**

5.1 Summary 53
5.2 Future Work 54
5.2.1 Expanding the Scope of the Property Language 54
5.2.2 Equivalence Checking 54

5.2.3 Interpolation on the Output Values 54
REFERENCES **56**

LIST OF FIGURES

2.1 LEMA block diagram. 9
2.2 Model generation example. 14
2.3 Phase interpolator model. 15
2.4 Phase interpolator model with pseudo transitions. 15
2.5 Phase interpolator model using the functional approach. 16
2.6 Phase interpolator model with merged transitions. 17
2.7 Property LPN for a phase interpolator. 18
3.1 An LPN model for the phase interpolator property. 20
3.2 LPN for *wait(b)*. 24
3.3 LPN for *waitPosedge(b)*. 25
3.4 LPN for *wait(b,d)* 26
3.5 LPN for *assert(b,d)*. 26
3.6 LPNfor *assertUntil(b1,b2)*. 27
3.7 LPN for *ifelse* function. 28
3.8 LPN for *example 1*. 31
3.9 LPN for *example 2*. 32
3.10 LPN for *example 3*. 34
3.11 LPN for *phase interpolator property*. 36
3.12 Phase Interpolator model to verify. 37
4.1 Block diagram for a phase locked loop. 41
4.2 A digital PLL design. 43
4.3 A simulation trace for a VCO. 43
4.4 An LPN model for VCO. 45
4.5 LPN model for *stable* generation. 47
4.6 An LPN model for VCO after adding *stable* variable. 48
4.7 Functional VCO LPN model that accounts for transient behavior. 50

ACKNOWLEDGEMENTS

The last two years at the University of Utah has been one of the most enriching experiences for me. My advisor and mentor, Dr. Chris Myers, has played a pivotal role in making this journey a success. Working with Chris has been a pleasant learning experience throughout. As a mentor, Chris did help me overcome all the technical difficulties that I faced during the course of the research. His patience and attitude gave me a great support. Along with the technical aspects of the work, Chris also helped me improve my writing skills. I am very grateful to Chris for all the support I received from him as his student.

I would like to thank Dr. Ken Stevens and Dr. Scott Little as my committee members and also for their valuable suggestions in this research. I would like to thank Scott for his input and help in carrying this research forward. I have bugged Scott with my questions many times and at all these times, Scott has patiently replied to my emails with in-depth explanation.

I would like to thank Semiconductor Research Corporation (SRC), National Science Foundation, and Intel Corporation for funding this research.

I am thankful to my family for their support, blessings, and confidence in me. I want to thank my friend Shomit Das for his help at different phases of my graduate studies. I would like to thank all my lab mates, especially Zhen, Andrew, Curtis, and Nic for making this a wonderful place to work and also for their help in all the small and big matters in the lab and in my work. I would like to thank Satish Batchu for helping me ramp up my understanding of the project after I took over his work.

CHAPTER 1

INTRODUCTION

The last decade has seen a tremendous growth in wireless and communication devices. This increasing consumer demand made circuits grow in complexity as they shrunk in size. In spite of being called a 'digital age', many of the systems used today have analog/RF components in them because real-world signals are analog in nature. Circuits get complex because of increased functionalities and performance requirements. To make sure circuits perform their intended functions, validation is important. SPICE simulation is still widely used for the verification of analog circuits. Although accurate, it gets very slow for complete functional verification of the circuit.

Boolean design, as well as simulation and verification, can be done efficiently with automated tools. However, the continuous nature of analog signals make it difficult to automate these processes for analog circuits. Thus, analog circuits have custom design processes which are very time consuming. The increased performance requirements these days have made this even more difficult. Shrinking of circuit size which is the direct effect of reduced transistor sizes is a boon for the digital circuit industry as it reduces the power consumption and increases the speed. However, it did not work as well with the analog industry as it created challenges such as reduced headroom available at the output, output impedance mismatch, process variation, and increased noise. Thus, design and validation has become even more daunting. For example, circuits like analog filters and analog *phase locked loops* (PLLs) are very difficult to design and verify, always leaving some room for error. To take advantage of automated tools available for digital circuits, a number of these inherently analog circuits are replaced by their digital counterparts. There now are digital filters, PLLs, etc. Although this solves some problems, it brings some other challenges. Inclusion

of digital blocks in analog circuits makes the verification at the interface of the analog and digital blocks difficult. Verilog-AMS and VHDL-AMS models of the circuits may be used for simulation and verification. In this case, input conditions are provided to the abstract model, and it is then simulated for those input values. Output is then checked to see if it meets the functional requirements. Although more efficient than SPICE simulations, it has some shortcomings. This method gets slower for big and complex circuits. Also, as the input conditions are provided manually, circuits might not get simulated for all the critical cases. Finally, these HDL models may not be accurate.

Formal verification methods consider all the possible input conditions and possible states of the system and generate a state-space for the system. These techniques have the potential to be a better way of validating the functionality of complex AMS circuits. Therefore, various efforts have been made to devise new methods of modeling AMS circuits and verify their correctness [1]. These models can be checked against verification properties to ensure that the circuit is correct. Formal verification can be grouped into *theorem proving* and *state space exploration* methods. In theorem proving, a designer generates a mathematical proof that the specifications of the circuit are met by its model. This makes verification easier at different hierarchical levels. Work done in [2, 3, 4] emphasizes theorem proving methods for verification of analog circuits, but these methods require a lot of expertise on the part of the designer, as well as the user.

In state-space exploration methods, the system is modeled as a state-space and then checked for all the inputs for all the possible states. It brings the advantage of automating the verification process, but it faces the problem of state-space explosion. Thus, it is essential to have a tool that can model the circuit at the right level of abstraction while maintaining accuracy. State-space exploration is further divided into *equivalence checking* and *model checking*. In equivalence checking, two models of a circuit are analyzed to check if they are functionally equivalent. This approach makes it possible to compare circuits at the same level, as well as at different levels of abstraction. In [5], the authors present an equivalence checking method for formal verification of linear analog circuits by representing the circuit's transfer function in

the 's' domain. Here, parameter variation is also taken into account. While this analysis can be applied only to linear circuits, in [6], the authors explain a method for time domain analysis of nonlinear circuits.

Model checking methods are useful for the verification of dynamic properties of the system. First, the circuit is modeled at the right level of abstraction and then reachability analysis is done to check if a certain state is reachable in the whole state space of the circuit. The model checking method is used in [7] for verifying nonlinear analog circuits. The authors discretized the continuous state-space to create an abstraction and represent the characteristics of transistors in multidimensional cubes. Conventional model checking algorithms are then used to verify the circuit. Polyhedral-based techniques are used in [8, 9, 10, 11, 12] to overcome the problem of computational complexity of the previous approach. In [13], the authors propose a *bounded model checking* tool for the verification of quasi-static behavior of AMS circuits using SAT modulo theories. However, the use of real numbers instead of rational numbers compromises the accuracy. *Run-time verification* methods are described in [14, 15] to eliminate the problem of state-space explosion. Here, logical monitors are developed for run-time verification of real-time, hybrid systems. A test-bench is attached to the circuit to be verified and is simulated. The assertions in the model are checked during the simulation; thus, there is no need for a circuit model before verification. Although interesting and simple, it faces many challenges, such as synthesizing accurate monitors from specifications and writing frequency domain properties. Dastidar et al. in [16] generate the FSM for the system using SPICE simulation traces. Here, the continuous state-space is discretized which is then converted to an acyclic, time-bounded FSM.

1.1 Model Generation

The key of successful verification lies in efficient modeling of circuits. A lot of work is being done to model circuits to capture the behavior correctly. Various techniques for abstracting linear systems have shown promising results [17, 18]. In [19], authors have explained *asymptotic waveform generation* method to model linear circuits like interconnect RLC models. The approach in [11] creates finite-state abstractions of

continuous analog behavior. Despite all this, modeling behavior of nonlinear circuits is a problem far from solved. The approaches used for modeling nonlinear systems rely on approximating them as either *piecewise linear* or *picewise polynomial* and then applying the abstract modeling techniques of the linear or weakly nonlinear systems [20]. In [21], the authors have proposed a method to model nonlinear circuits where they represent these circuits as piecewise linear systems and then reduce these pieces. In [22], the authors build the macro-models of the nonlinear circuits. It is done by catching the trajectory of the states that the circuit goes through and then using linearization among the time points in between. In [23], authors present a compact nonlinear model order-reduction method (NORM). It is suitable for model order reduction of a class of weakly nonlinear systems that can be well characterized by low-order Volterra functional series. In [24], the authors talk about the need of hierarchical modeling of AMS circuits and propose algorithmic techniques for automatically extracting a suitable nonlinear macro-model from a device-level circuit. Here, a circuit is represented by a small set of mathematical equations extracted from a large set of mathematical equations that model the circuit. Dastidar et al. in [16] generate the FSM for the system using SPICE simulation traces. Here, the continuous state-space is discretized which is then converted to an acyclic, time-bounded FSM.

LPN Embedded Mixed-Signal Analyzer (`LEMA`) [25, 26] is being developed for formal verification of AMS circuits. This tool supports a *simulation aided verification* methodology where the simulation traces generated by SPICE level simulation of AMS circuits are used as an input to the tool. The tool generates a *labeled Petri net* (LPN) model for the circuit. This model can be checked against the property given as input to the tool. Here, the input trace is discretized and a state-space is generated. The generated model contains information present in the simulation trace input. Thus, the model generated is specific to the input values considered during simulation. A more generalized form of the model is necessary to make sure the model works when subjected to different input conditions during verification.

1.2 Property Generation

Having a good model of the circuit or system is not enough. If the property cannot express the expected behavior efficiently, model checking cannot be advantageous. If the circuit is modeled in HDL languages, then it is customary to have the properties written in HDL languages too as it makes the verification easy. Temporal logic is useful in defining the properties over time and thus important for specifying AMS properties. In [27], the authors describe monitoring algorithms for checking temporal properties of discrete, timed, and continuous behaviors. *Property Specification Language* (PSL) is a formal specification language for specifying properties of both digital and AMS circuits. In [28], the authors used PSL to express temporal properties of the AMS circuits. In [29], the authors apply a property-based checking methodology where the circuit behavior is expressed in PSL in the form of assertions. The methodology is applied to the simulation traces generated from practical DDR2 interface design. In [30], authors talk about modeling the AMS design in terms of a *System of Recurrence Equations* (SRE). Then, an assertion-based verification method is defined using the symbolic trace of SRE. For the verification of static and dynamic properties of analog systems, the authors in [31] present a new *Analog Specification Language* (ASL) to specify the properties efficiently. In [32], the authors describe writing properties to handle the asynchronous behavior inherent in synchronous circuits. In [33] , the authors discuss a variant of temporal logic tailored for specifying desired properties of continuous signals. In [29], the authors discuss automatic construction of an observer from the specification in the form of a program that can be interfaced with a simulator. In [34], the authors have proposed a new syntax and semantics for real-time regular expressions, which extend existing *SystemVerilog assertion* (SVA) regular expressions. In [35], the authors describe a methodology for dynamically verifying complex AMS properties. These properties are encoded with the help of local variables and mapped into SVA properties.

1.3 Contributions

Verification is done by checking the generated model against the property that the circuit is supposed to satisfy. This property is currently encoded using LPNs.

Building properties in this way is a tedious process, and it gets even more complex for an AMS circuit like a PLL. Also, it requires the user to have a considerable familiarity with the tool. To solve this problem, we have developed a new property language so that it is easy to write the properties for AMS circuits. However, as the verification properties in LEMA have to be in LPN format, it is important to convert these properties in LPN. To enable this, we have created a translator that converts a new property specification language into LPN models.

This thesis also describes new methods that improve and generalize our model generation method. Whenever an input to the circuit is changed, the output changes but does not settle down to the expected value immediately. Instead, it goes through a transient period during which its value is unpredictable. This thesis contributes a method to model this transient period, thus separating it from the stable output period.

Our tool takes SPICE simulation traces of the circuits as input for model generation. Therefore, the circuit is modeled only for the input values considered during simulation of the circuit. To make the model represent the circuit behavior for a wide range of input values, we have implemented a type of linear abstraction, *interpolation*. This work assumes that though analog circuits are considered nonlinear, it is always possible to convert a circuit to a set of variables where the linear abstraction holds [36].

Thus, the important contributions of this thesis are,

- Development of a new property language and a translation method to convert these properties to LPN monitors.
- Improved model generation by separating transient period from stable period.
- Further generalization of AMS models using a linear abstraction.

1.4 Thesis Overview

This thesis is divided into five chapters. Chapter 2 gives a detailed description of the design and working of the tool `LEMA`. It also introduces LPNs and describes their usefulness in modeling AMS circuits. Chapter 3 and Chapter 4 describe the contributions that this thesis has made. Chapter 3 highlights the need of the new

property language for verification. It details the new language developed, and its conversion to LPN properties. Chapter 4 deals with the improvements made in generating models by separating the transient period of the circuit operation from the steady state. It also states the need for generalization of models and discusses *interpolation* as a means to achieve it.

CHAPTER 2

BACKGROUND

This chapter introduces the the tool *LEMA*. Section 2.1 describes the LEMA functional flow. Section 2.2 then describes LPN syntax and semantics. Section 2.3 describes the LPN generation process. Finally, Section 2.4 describes the verification property to be used with the generated models to check the correctness of the circuit.

2.1 LEMA

The enormous tool support that digital circuits have received has brought about a revolution in the way digital circuits are synthesized, simulated, and verified. Because of the basic Boolean nature of digital circuits, it has become possible to automate these design steps. Verilog or VHDL models of digital circuits are used with these tools to generate gate-level and transistor-level circuits. Layout is then built for these circuits. Circuit and layout is checked for errors. Simulation and verification is automated too. While simulating the circuit for a particular set of inputs, a set of expected outputs can be stored and the simulation output is compared with it to check the correctness. Such support for AMS circuits is not available. Thus, design is still a custom process and simulation is still a widely used method for verification. It is very tricky to abstract an AMS circuit into a HDL model as the functionalities of these circuits are complex.

LEMA supports a *simulation aided verification* (SAV) methodology to generate the abstract models for AMS circuits. Fig. 2.1 shows the block diagram of `LEMA`. This tool takes as input the simulation traces generated by SPICE level simulation of the circuit. These traces are compiled by our model generator into an abstract SystemVerilog model for the circuit. This model can then be combined with an RTL model for the digital components for system-level simulation. Our model genera-

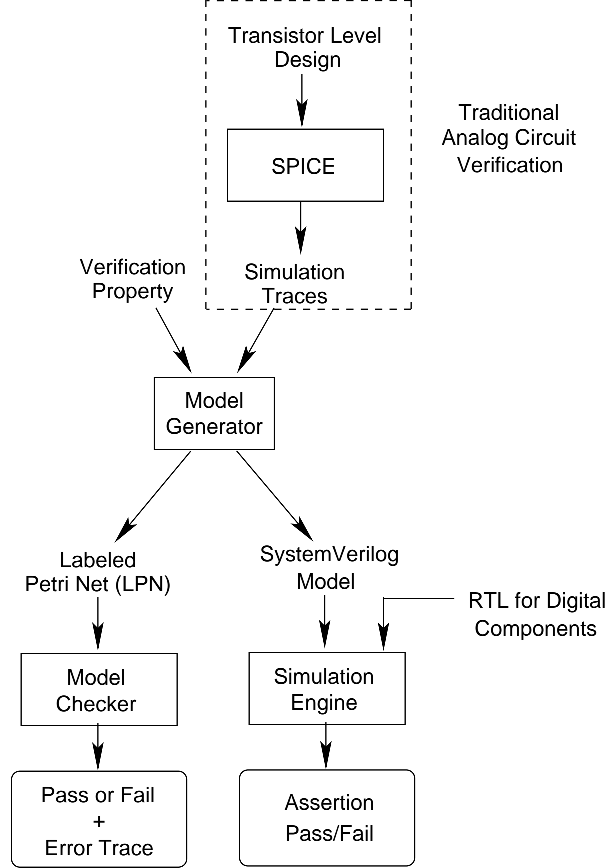


Figure 2.1: LEMA block diagram.

tor also allows the user to provide a verification property which is converted into SystemVerilog assertions. These assertions can then be checked during simulation. Finally, our model generator can generate a formal model of the circuit in the form of a LPN [26, 37]. This LPN which includes the verification property can be analyzed by our model checker described below. The model checker checks that the verification property holds under all variations allowed by the model. If the property fails, an error trace is generated which can be analyzed to find possible errors in either the circuit or the abstract model.

LEMA includes three model checkers which differ in how they represent and explore the state space. The first uses *zones* represented with *difference bound matrices* (DBM) [39]. This method represents the infinite state-space within a finite number of state equivalence classes called *state sets*. The continuous portion of the state-space is represented by different zones, using matrices of inequalities called

DBMs. These inequalities are of the form $x_i - x_j \leq k_{ij}$, where x_i and x_j are the values of a transition's clock or the continuous variables and k_{ij} is a constant or ∞, if it is unbounded. DBMs are a restricted form of convex polygons that use only $45\,^\circ$ and $90\,^\circ$ angles which indicates that the continuous variables can change with rate one only. However, in `LEMA`, a technique called *warping* is used to represent continuous variables changing with rates other than one. Reachability analysis is then performed to check the model for fail transitions. During this analysis, if the property fails, an error trace is generated by the tool.

The second model checker uses *binary decision diagram* (BDD). For model checking using a BDD, the relationships between continuous variables are represented using separation predicates in a canonical form. These separation predicates are mapped to Boolean variables enabling the use of BDD operations to evolve the state. The algorithm begins with the complement of a *timed computation tree logic* (TCTL) property and the reachability analysis is done backwards. This process continues until either finding an initial state or the state representation does not change. If an initial state is reached, it indicates that a failure state can be reached.

The third model checker uses a *satisfiability modulo theories* (SMT) solver [38]. In the SMT-based model checker, the system model is transformed into an SMT formula composed of the initial state, transition relations, and assertions. As SMT is a *bounded model checking method*, the model is explored for a fixed number of iterations. In the first step, state variables are created for each iteration. The initial state is asserted and an SMT formula is constructed for the first iteration. Next, iteration state variables are calculated and finally, a failure condition is asserted in each iteration. If any of these assertions can be made true, it indicates the verification fails.

2.2 Labeled Petri Net (LPN)

A *Petri net* is essentially a state transition diagram. A LPN is a variant of Petri nets where transitions that connect two places have a labeling function. This extension makes it possible to model AMS circuits using LPNs. A LPN can model the continuous signals in AMS circuits that have varied rates of change. A LPN

has places and transitions. Places are the states and transitions connect these places indicating how the state can change. Transitions have an enabling condition, delay, and assignments. With the ability to add rates for continuous signals, LPNs can efficiently model AMS circuits. The following sections explain the syntax and semantics for LPNs.

2.2.1 LPN Syntax

An LPN is a tuple $N = \langle P, T, X, V, F, M_0, Q_0, R_0, L\rangle$[1]

- P : is a finite set of places;
- T : is a finite set of transitions;
- $T_f \subseteq T$: is a finite set of failure transitions;
- V : is a finite set of continuous variables ($V = V_i \cup V_o \cup V_n$);
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- $M_0 \subseteq P$ is the set of initially marked places;
- $Q_0 : V \rightarrow (\mathbb{Q} \cup \{-\infty\}) \times (\mathbb{Q} \cup \{\infty\})$ is the initial range of values for each continuous variable;
- $R_0 : \Delta \rightarrow (\mathbb{Q} \cup \{-\infty\}) \times (\mathbb{Q} \cup \{\infty\})$ is the initial range of rates of change for each continuous variable.
- L : is a tuple of labels defined below;

A LPN consists of a finite set of places, P, and a finite set of transitions, T. As the model is checked against the verification property, it has a finite set of failure transitions T_f. All these variables have an initial value and rate of change. The set V represents the set of continuous variables that are used to represent the signal values in the AMS circuit. The flow relation, F, describes how places and transitions are connected. The sets $M0$, $Q0$, and $R0$ represent the initial markings of the places, the initial values of the continuous variables, and the initial rates of change for the continuous variables, respectively.

[1] A somewhat simplified version of LPNs is used in this thesis which is sufficient for AMS circuit models.

Formal definition of the labels on a transition in an LPN is a tuple, $L = \langle En, D, VA, RA \rangle$:

- $En : T \rightarrow \mathcal{P}_\phi$ labels each transition $t \in T$ with an enabling condition.
- $D : T \rightarrow \mathcal{P}_\chi$ labels each transition $t \in T$ with a delay for which t has to be enabled, before it can fire.
- $VA : T \times V \rightarrow \mathcal{P}_\chi$ labels each transition $t \in T$ and continuous variable $v \in V$ with the continuous variable assignment that is made to v when t fires.
- $RA : T \times \Delta \rightarrow \mathcal{P}_\chi$ labels each transition $t \in T$ and continuous rate variable $v \in V$ with the rate assignment that is made to v when t fires.

The enabling conditions are Boolean expressions, $\mathcal{P}_\phi$, that satisfy the following grammar:

$$\phi \ ::= \ \mathbf{true} \mid \neg\phi \mid \phi \wedge \phi \mid v_i \geq c_i$$

where $\neg$ is negation, $\wedge$ is conjunction, v_i is a continuous variable, and c_i is a rational constant. The assignments are numerical formula, ϕ_e, that satisfy the following grammar:

$$\chi \ ::= c_i \mid \infty \mid v_i \mid (\chi) \mid -\chi \mid \chi + \chi \mid \chi * \chi \mid \mathrm{INT}(\phi) \mid \mathrm{uniform}(c_i, c_i) \mid rate(v_i)$$

where the function $\mathrm{INT}(\phi)$ converts a Boolean **true** or **false** to 1 or 0, respectively. The function $\mathrm{uniform}(l, u)$ gives a uniform random value in the interval (l, u), and the function $\mathrm{rate}(v_i)$ returns the current range of rates for the continuous variable v_i.

2.2.2 LPN Semantics

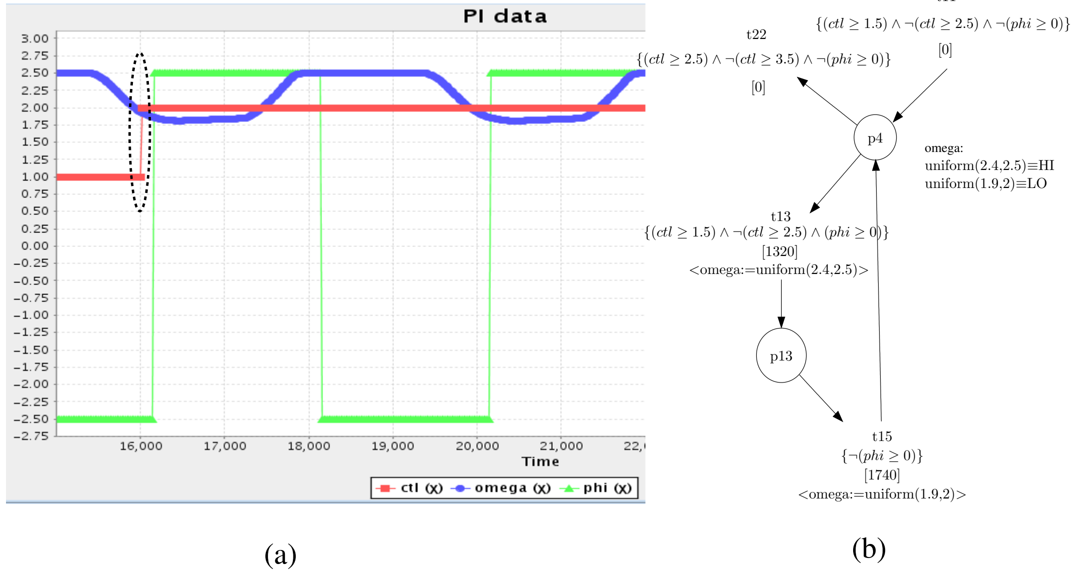
The current state of a LPN is the set of places which contain a token, the current value of all discrete and continuous variables, and the current rate of change of each continuous variable. A place with a token in it indicates the current state of the system. A transition fires when its preceding place has a token, and its enabling condition is satisfied. An enabling condition is one of the labels of the transitions in the LPN. When a transition fires, after the delay period labeled on the transition, the token moves to the next place and the assignments to the variables in the system

are made.

Consider the portion of the simulation trace of a phase interpolator circuit shown in Fig. 2.2(a). The LPN that LEMA generates for this portion of the trace is shown in Fig. 2.2(b). The highlighted part shows that the control input changes from 1 to 2 V while the input clock is less than 0. This change is captured by the transition $t11$. For this transition, the enabling condition encodes the fact that it should fire when *ctl* is between 1.5 and 2.5 volts and the input *clock* is less than 0 volts. Note that our model uses ranges rather than exact values since in analog circuits, the values may be noisy. The delay of this transition is 0, indicating that the state change happens immediately. Therefore, after $t11$ fires, the state changes to place $p4$. When the system is in state $p4$, *ctl* remains the same but the input clock *phi* can change from low to high. This transition enables a change in the output clock *omega*. However, this output change does not take place immediately, so the transition is given a delay of 1380 time units to reflect the time until the output changes. After this delay, the transition fires, and when it does, the variable assignment sets *omega* to a random value uniformly distributed between 2.4 and 2.5 volts to capture possible noise. The firing of this transition, $t13$, takes the system to the state $p13$. In this state, the input *clock* changes back to a low value, and it excites a change in *omega* to a value of 1.9 to 2 volts after a delay of 1740 time units. This firing also takes us back to the state $p4$ where the circuit waits for the next rising transition on the input *clock*.

2.3 LPN Generation Process

This section describes how the LEMA tool generates an LPN from simulation output files. These output files can be in one of the .dat, .csv, or .tsd formats. `LEMA` extracts the information about all the input and output signals. The variables can be set as *discrete multivalue* (DMV) or continuous. DMV variables are the continuous variables that are constant for most of the time. The variables that are constant for a predetermined percentage of time of total trace period, are set as DMV. The variables that are not DMV are set as continuous. The whole trace is then divided into regions and each of these regions is represented with a place in the generated LPN. These regions are set by using *thresholds* on each of the variables.



(a) (b)

Figure 2.2: Model generation example.

Fig. 2.3 shows the complete LPN generated for a simulation trace in which the control input starts at 1V, after some time changes to 2V, and finally, changes one last time to 3V. There is a problem with this model though. Namely, it has encoded the input sequence used by the circuit designer during simulation. If this model is subjected to a different input sequence, it may show unusual behavior. For example, if the input sequence changes from 1V to 3V, then the output continues to show the phase shift for the 1V control value since there is no transition to take the system from the 1V mode to the 3V mode directly.

To address this issue, pseudo transitions can be introduced as shown in Fig. 2.4. These pseudo transitions are added between the places to make sure the system does not get stuck in an incorrect state and give erroneous behavior. As shown in Fig. 2.4, transition *pt1* takes the system from *ctl* value of 1 to 3, solving the previous problem. Similarly, *pt0* allows the system to transition from 2 to 1, *pt2* takes the system from 3 to 1, etc. It should be clear though that pseudo transitions can add substantial complexity to the generated models.

To address this problem, the model generation method within LEMA can also use a functional approach. This introduces a notion of *care* variables, and it only introduces a new state in the model when there has been a change in a *care* variable.

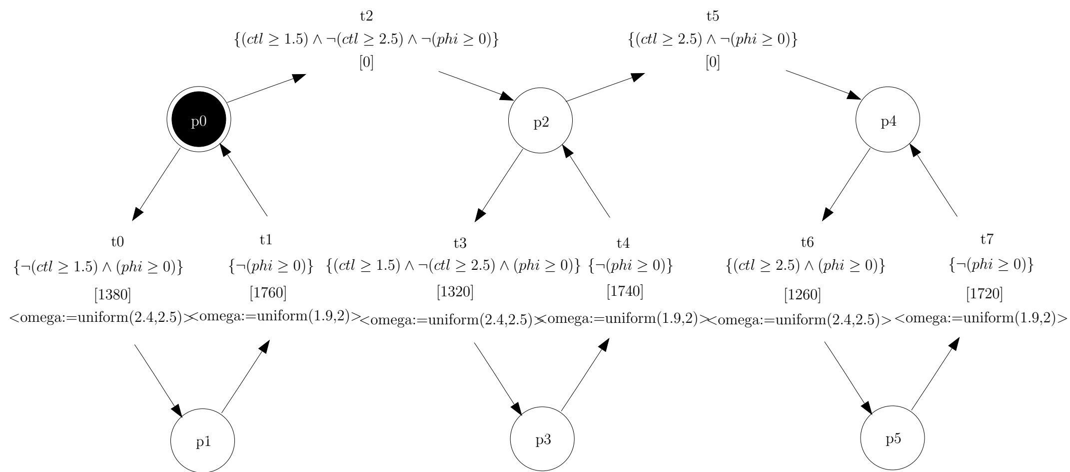


Figure 2.3: Phase interpolator model.

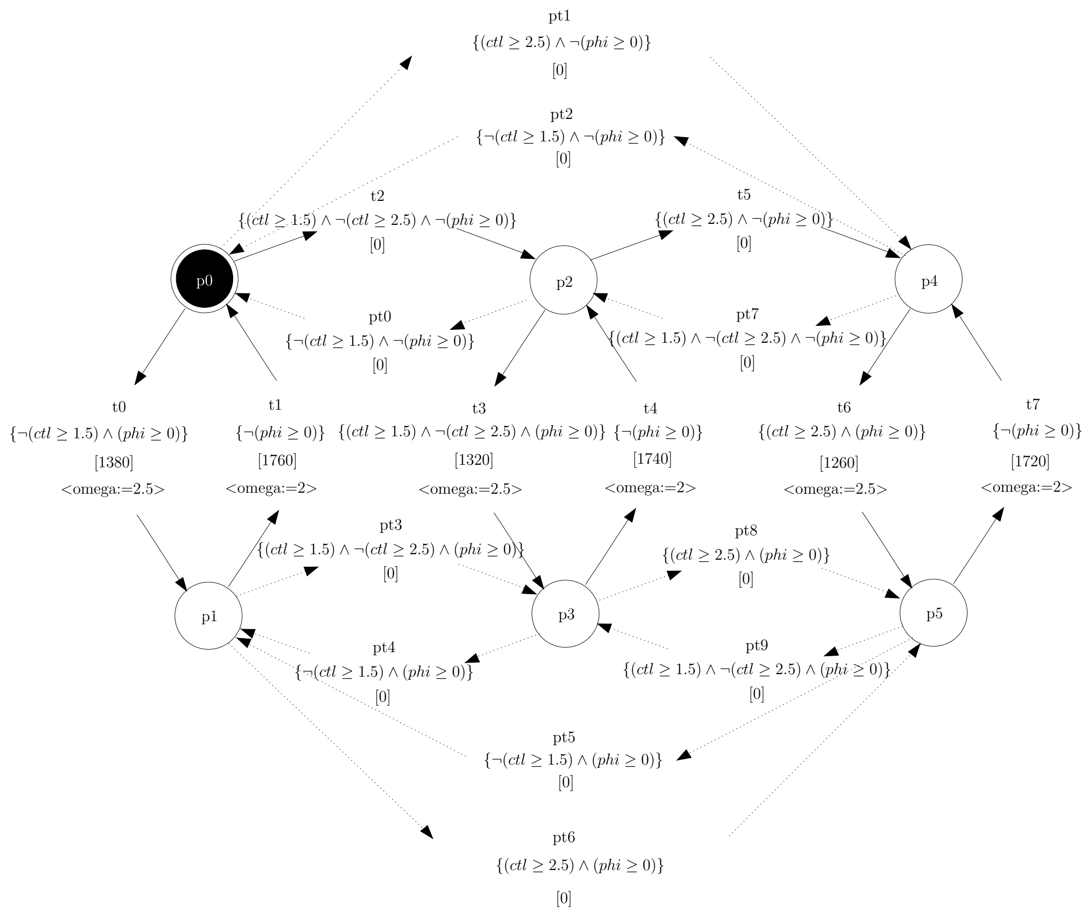


Figure 2.4: Phase interpolator model with pseudo transitions.

This example has only the output *omega* as a *care* variable. Therefore, the changes in the input values are not recorded in a state. This reduces the number of places from 6 to 2, and it also dramatically reduces the number of transitions as pseudo transitions are no longer required to encode alternative input sequences. The intuition is that the order of the input changes is no longer encoded in the model. The generated model is shown in Fig. 2.5.

This model can also be simplified a bit more by observing that many transitions only differ in their enabling condition and delay. These transitions can be merged as shown in Fig. 2.6. This final reduced model now has a single transition to set the output omega high with the delay of this transition computed as a function of the control input. Similarly, there is a single transition created for setting the output low. Indeed, it should be clear that this automatically generated model is exactly what one would expect of a phase interpolator. Namely, when the input clock changes, the output clock changes after a delay determined by the value of the control input.

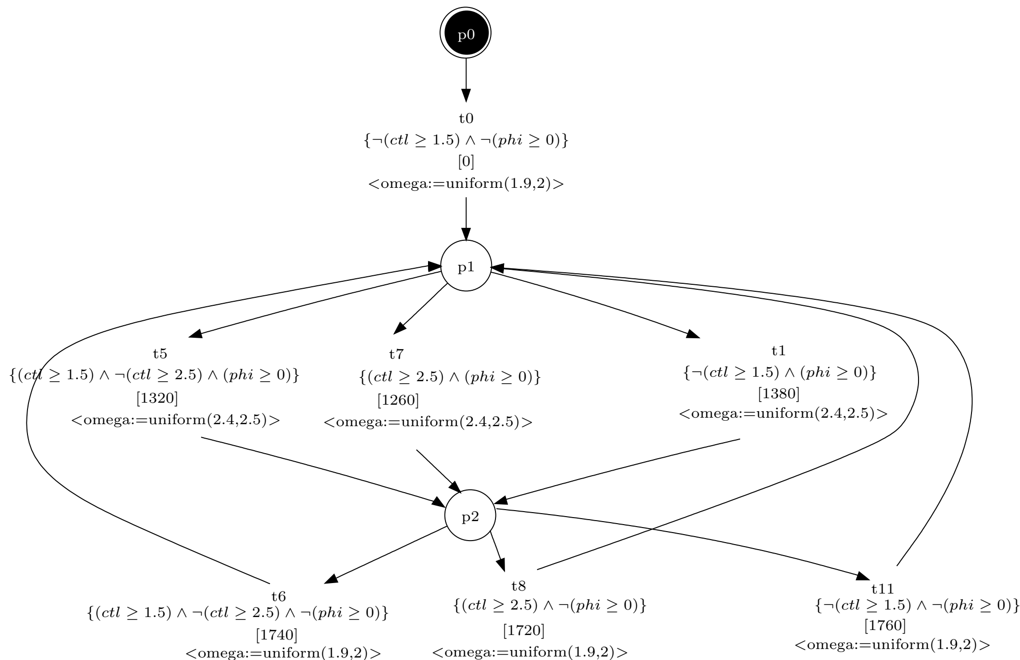


Figure 2.5: Phase interpolator model using the functional approach.

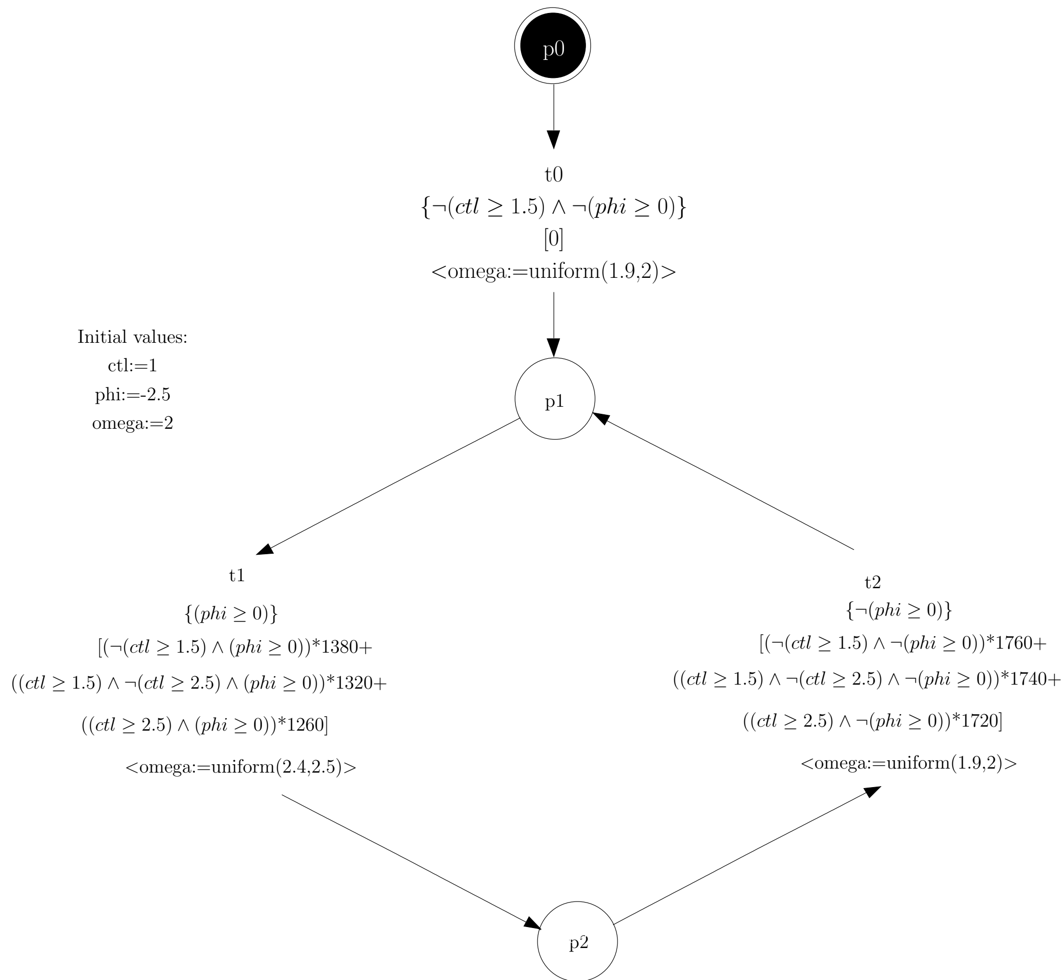


Figure 2.6: Phase interpolator model with merged transitions.

2.4 Verification Property

The properties to be verified for these models are currently encoded using LPNs. Fig. 2.7 shows the property LPN for a phase interpolator circuit. This property is verified under all the variations allowed by the model generated for the circuit. In this LPN, transitions *tFailMin, tMax1, tMax2*, and *tMax3* are fail transitions. During verification, if any of these transitions fire, the tool records the failure and generates an error trace. This error trace is then analyzed by the designer to find the bugs in the circuit. This model waits for the clock to go high, it then puts tokens in the 2 places, $pCheckmin$ and $pCheckMax$. The place $pCheckMin$ is used to check for a minimum phase delay on the output clock while the place $pCheckMax$ checks for a maximum phase delay on the output clock. For example, if control has a value of 1,

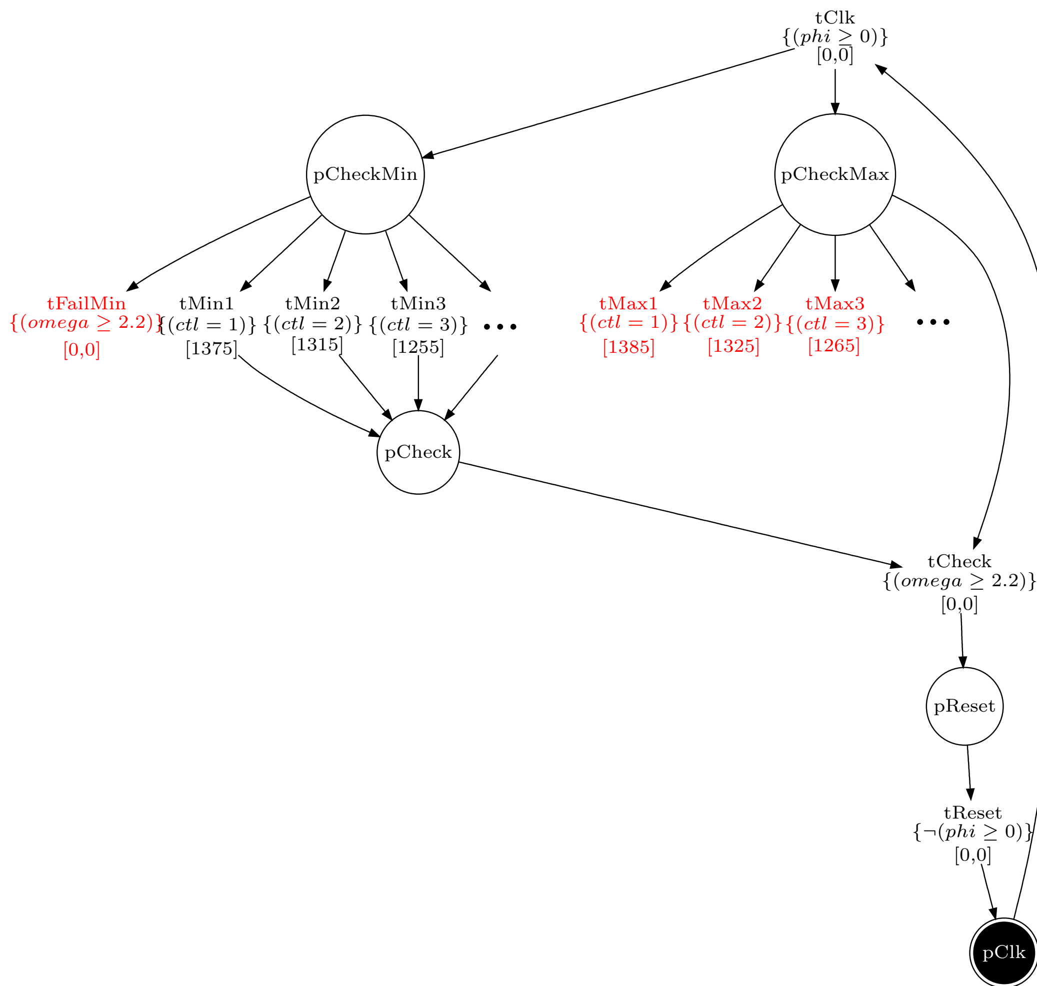


Figure 2.7: Property LPN for a phase interpolator.

the phase delay should be between 1375 and 1385. While $pCheckMin$ is marked, if $omega$ goes high before $tMin1$ fires, then $tFailMin$ fires, indicating a minimum delay failure. If 1375 time units pass without $omega$ changing, the token from $pCheckMin$ moves to $pCheck$. Now, it is expecting a change in $omega$. However, if 1385 time units pass since the input $clock$ before $omega$ changes, the transition $tMax1$ fires indicating a maximum delay failure. If $omega$ does fire in time, $tCheck$ fires moving the model to the $pReset$ state where it waits for the input clock to go low and high again before checking the property again.

CHAPTER 3

PROPERTY LANGUAGE TRANSLATOR

This chapter talks about the development of a new property language and its conversion to LPNs. Section 3.1 talks about the need for the new language for property specification. Section 3.2 talks about the use of SVA and *real-time SVA* (RT-SVA) for property specification. It also explains the limitations of SVA and complexity of RT-SVA for specifying AMS properties. Section 3.3 describes the contributions of this thesis. It explains the new property language functions developed as a part of this thesis and their conversion to LPNs. Section 3.4 explains the compiler developed for parsing the input properties. Further, Section 3.5 demonstrates the use of these property functions to express some AMS properties.

3.1 Motivation

As discussed previously, model checking is a type of formal verification that is used extensively for AMS circuits. In model checking, first, a model for the circuit is built. This model is then checked against the properties that the circuit is supposed to satisfy. If the circuit fails to satisfy this property, verification fails. LEMA implements model checking. It generates the LPN models for the circuits. These models are then checked against the property. These properties are also expressed in LPN format.

Fig. 3.1 shows the property for verifying the correctness of the phase interpolator circuit. The property is built manually. Taking a look at the property LPN, it is clear that building LPN properties manually is a tedious and time-sconsuming task. Therefore, a translator that can convert important AMS properties into LPNs would be helpful.

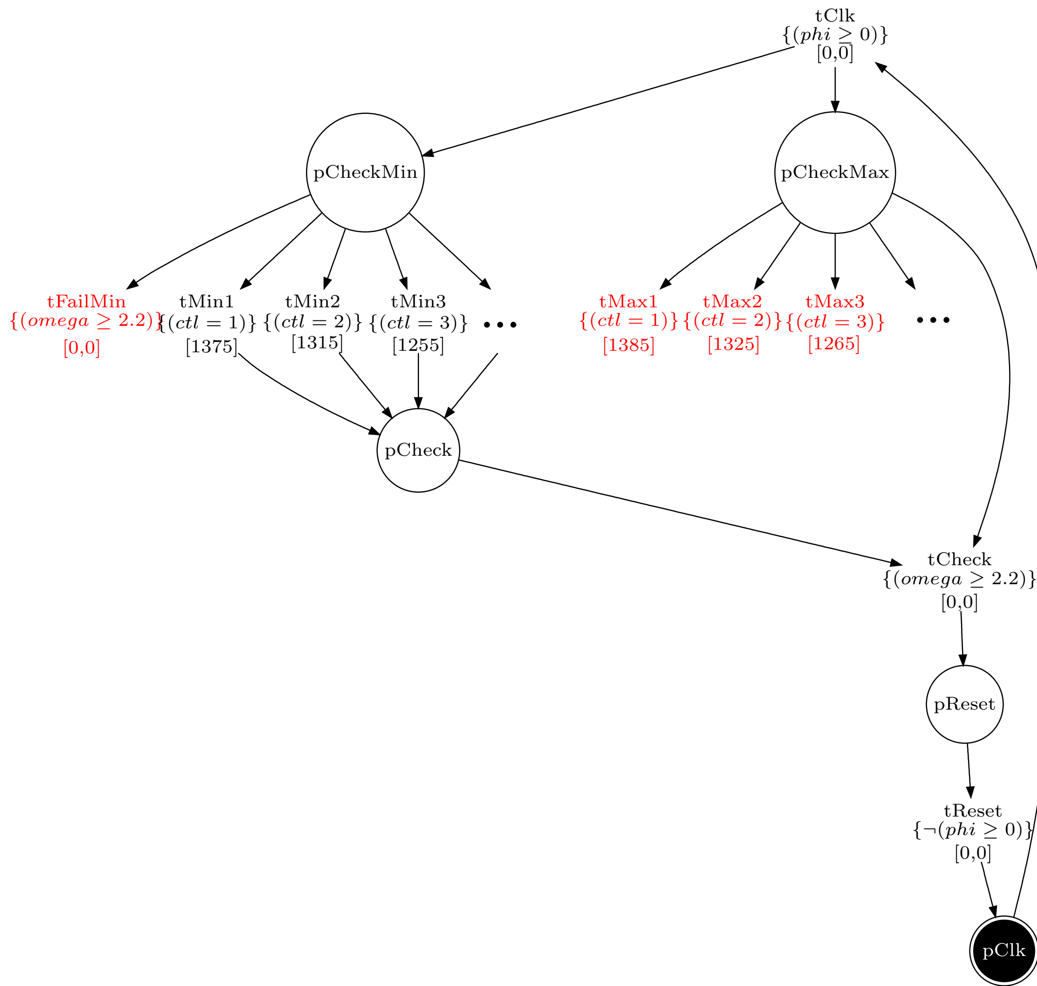


Figure 3.1: An LPN model for the phase interpolator property.

3.2 Real-Time SVA

Assertions are a useful way to validate the circuit functionality. They can be used during simulation, as well as the formal verification of the circuit. In simulation methods, the property is a part of the behavioral model of the circuit and is checked during simulation. Assertions can also be checked formally using model checking.

SVA can describe the properties of circuits over time. Thus, it is useful in writing temporal properties of a circuit. Also, familiarity of designers and verification engineers with SVA makes it a perfect choice for writing properties. Shown below are a couple of sample SVA properties.

```
assert (a == b);
assert property (@(posedge clock) req |−> ## [10:20] ack);
```

The first is an example of an immediate assertion where it is just checked that 'a' is equal to 'a'. The second is an example of a concurrent assertion. It checks that at the positive edge of the clock, if request signal is high, the *ack* signal should go high within 10 to 20 clock cycles.

There are problems with writing properties of AMS circuits using SVA. As we can see, the concurrent SVA can be checked only on an occurrence of an event. Here, the number of clock cycles is a parameter used to describe the properties over time. Thus, SVA is useful for writing properties of synchronous digital circuits. However, in the case of AMS circuits, this synchronous clock is absent. Therefore, writing temporal properties for AMS circuits becomes difficult. This problem can be solved by introducing a pseudo clock in the circuit model and checking the assertion at the positive edge of this pseudo clock. However, the continuous nature of all the signals can produce unpredictable results.

To address this problem, the authors in [34] have proposed real-time regular expressions as an extension to the existing SVA regular expressions. These RT-SVA expressions use discrete-time semantics for digital sequences and real-time semantics useful for AMS circuit verification. Real-time sequences are specified using the grammar below:

R ::= @(κ)(b) | R ##1 R' | R ##0 R' | R or R' | R intersect R' | R[*0] | R[+] | b | b[*α [+] : β [-]]

Here,

- κ is an event and b is a Boolean expression. @(κ)(b) specifies that b is true at a point near to κ.
- ##1 and ##0 are the nonoverlapping and overlapping concatenation operators. Thus, R ##1 R' specifies that R is satisfied followed by R' being satisfied. They

do not need to be satisfied at the same time instant while R ##0 R' specifies that there is an instant in time where they are both satisfied.

- *or* is the union operator. R or R' specifies that in an interval, either R or R' is satisfied.
- *intersect* is the intersection operator. R intersect R' specifies that in an interval, R and R' are both satisfied.
- [∗0] specifies zero repetitions. Thus, R[*0] specifies that R is empty.
- [+] specifies one or more repetitions.
- α [+] indicates an instant of time greater than amount α and β [-] indicates an instant of time just less than β. Thus, b[*α [+] : β [-]] specifies that b is true between α and β time units.

These basic sequences are used to derive other sequences such that its application to practical AMS circuits becomes easy. Shown below is the phase interpolator property expressed using RT-SVA:

$$
\begin{gathered}
(phi \geq 0)[\sim\!> 1] \;\#\#0 \\
(((ctl == 1)[\sim\!> 1] \;\#\#0 \\
(!(omega > 2.2))[*1375 : 1385] \;\#\#1\; (omega > 2.2)) \\
or \\
((ctl == 2)[\sim\!> 1] \;\#\#0 \\
(!(omega > 2.2))[*1315 : 1325] \;\#\#1\; (omega > 2.2]) \\
or \\
((ctl == 3)[\sim\!> 1] \;\#\#0 \\
(!(omega > 2.2))[*1255 : 1265] \;\#\#1\; (omega > 2.2))) \;\#\#1 \\
(phi < 0)[\sim\!> 1]
\end{gathered}
$$

While RT-SVA does allow AMS properties such as those in [40] to be specified, it can be tricky and somewhat tedious to read and write these verification properties in RT-SVA.

3.3 Property Language

To make the properties easy to read and write, this thesis proposes a new property language which is then converted to a property LPN automatically. In LEMA, the property is given as an input file with a *.prop* extension. The Listing 3.1 below is the format of this property file. The property starts with the property name. It is then followed by the variable declaration. Variables can be of two types; Boolean and real. After this, the property is specified in an *always* block. The always block here implies that the property statements in this block are checked repeatedly, i.e., after the property is finished checking once, the state returns to the beginning, enabling the verification of the property again. Note that this does not enable the threaded execution of the property.

```
property <name>{
  <declarations>
  always{
    <statements>
  }
}
```

Listing 3.1: Format for a property in new language.

Examples below show the property functions and their conversion to RT-SVA and property LPN.

1. **wait(b)**

   Here, wait(b) waits for b to be true. Here, b is a Boolean expression or a Boolean signal. LPN translation of the same is as shown in Fig. 3.2. Here, transition *t0* fires when b becomes true. There is no time limit; *t0* waits as long as necessary for b to become true. The RT-SVA translation for this is, $b[\sim> 1]$. This is the derived form of the basic syntax, $!b[*0:\$]\ \#\#1\ b$. Here, the first part matches the sequence where b is not true for any time between 0 and infinity and the sequence after that where b becomes true.

2. **waitPosedge(b)**

   waitPosedge(b) waits for the positive edge of expression b. LPN translation of

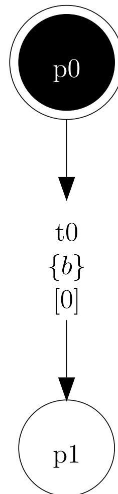


Figure 3.2: An LPN conversion for the *wait(b)* statement.

the same is as shown in Fig. 3.3. The RT-SVA translation for this is, $!b[\sim> 1]$ ##1 $b[\sim> 1]$. Note that this is equivalent to wait($\sim$ b) followed by wait(b).

3. **wait(b,d)**

   wait(b,d) waits for d time units for b to be true. Here, b is a Boolean expression. LPN translation of the same is as shown in Fig. 3.4. If b is false initially, the failure transition's enabling condition is satisfied, but it has a delay of d time units. If in this time interval, b goes true anytime, *t0* is fired and as it has 0 delay, it is fired immediately. If b does not become true, *tFail0* is fired and a failure is recorded. The RT-SVA translation for this is, $!b[*0 : d]$ ##1 b. Here, the first expression checks that b is false for sometime between 0 and d. The sequence then concatenates this expression with the one where b is true.

4. **assert(b,d)**

   assert(b,d) asserts that the expression b is true for d time units. Here, b is a Boolean expression or a Boolean signal. b has to be true continuously for d time units. If b becomes true and then goes false within d time units, a failure is recorded. LPN translation of the same is as shown in Fig. 3.5. $t0$ fires when b becomes true after a delay of d time units. In the meantime, if b becomes

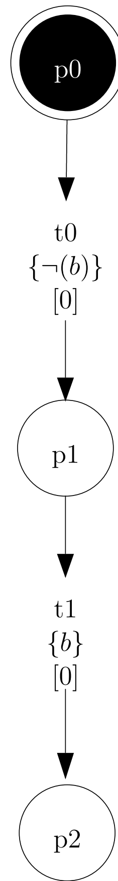


Figure 3.3: An LPN model for the *waitPosedge(b)* statement.

false, *tFail0* is fired. The RT-SVA translation for this is, $b[*d : d]$. Here, the expression checks that b is true for d time units exactly.

5. **assertUntil(b1,b2)**

   assertUntil(b1,b2) asserts that *b1* is true until *b2* becomes true. Here, *b1* and *b2* are Boolean expressions. LPN translation of the same is as shown in Fig. 3.6. Here, when d becomes true, $t0$ fires but if $b1$ becomes false when $b2$ is not true, a fail transition fires. The RT-SVA translation for this is, $b1$ & $!b2[*0.0 : \$]$ ##1 $b2$. This sequence checks that $b1$ is true when $b2$ is false and later $b2$ becomes true.

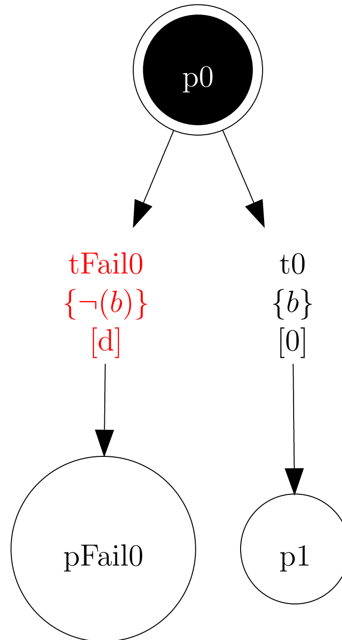


Figure 3.4: An LPN model for the *wait(b,d)* statement.

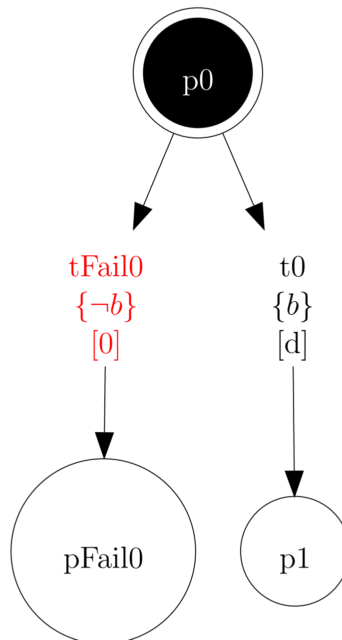


Figure 3.5: An LPN model for the *assert(b,d)* statement.

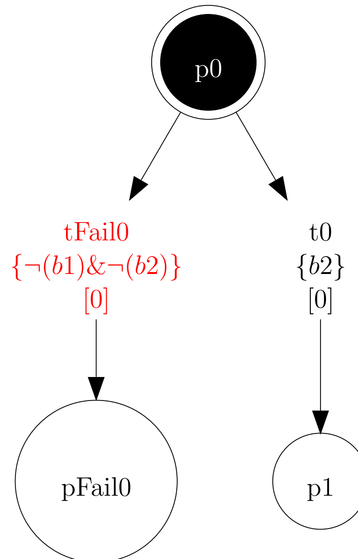


Figure 3.6: An LPN model for the *assertUntil(b1,b2)* statement.

6. Our property language also includes an ***if-elseif*** construct that be written as

   if (b1) {

   R1

   }***else if*** (b2) {

   R2

   } **else** {

   R3

   }

   which in RT-SVA is equivalent to :

   $b1$ ##0 $R1$

   or

   ($b2$ & !$b1$) ##0 $R2$

   or

   (!$b2$ & !$b1$) ##0 $R3$

   Fig. 3.7 shows the LPN conversion for the *if-elseif* function.

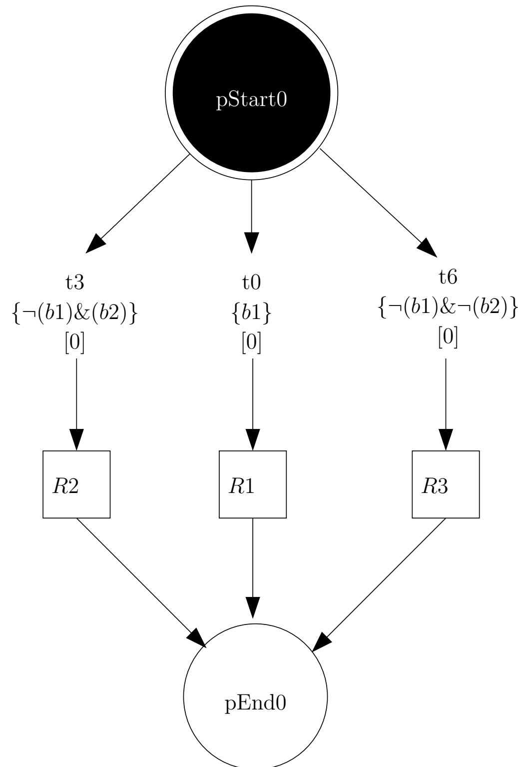


Figure 3.7: An LPN model for the *if-elseif* statement.

3.4 Property Compiler

The functions described above are applied to generate verification properties of AMS circuits. In a property where these functions are used together, LPNs are generated for these functions individually and then stitched together, thus producing the complete property net. A parser generator, *Another Tool for Language Recognition* (ANTLR) is implemented to automate this LPN generation in LEMA. ANTLR takes as input a grammar that specifies a language and generates as output the source code for a recognizer for that language. To perform any further actions upon the recognition of this grammar, these actions can be embedded within the code. In our case, these actions are producing output LPNs for the input grammar. Thus, after the property is given as input to LEMA, it gets parsed and if there is no syntax error with it, an LPN is generated for it.

The algorithm to generate the property LPN from the input .prop file is shown below.

1. Parse the name of the input property.
2. Parse the variable names and types. Add the variable names to the LPN variables list.
3. Parse the first statement, generate the LPN, put the token in its first place.
4. While parsing the next statement, merge the last place of the previous LPN with the first place of the next LPN.
5. Repeat 4 for all the statements in the property.
6. At the end of the *always* loop, connect the last place with the first place of the LPN, thus making a loop.

3.5 Examples

This section demonstrates the use of the property language to express the properties of AMS circuits. It also shows their equivalent RT-SVA expression form and LPN model generated automatically. The time to convert these propertes to LPN is less than a second.

1. Whenever a goes from zero to one, b remains low for at least 5 ms.

   This property can be expressed as shown in the Listing 3.2 below.

```
property example1 {
    boolean a;
    boolean b;
    always{
        waitPosedge(a);
        assert(~b,5);
    }
}
```

Listing 3.2: Example 1.

RT-SVA conversion for this property is as follows:

$!a[\sim> 1]\ \#\#1\ a[\sim> 1]\ \#\#1\ !b[*d : d]$

This gets translated to the LPN shown in Fig. 3.8.

2. After a goes high, b and c must intersect within 25 ns.

   This can be expressed as shown in the Listing 3.3 below.

```
property example2{
    boolean a;
    boolean b;
    boolean c;
    always{
        waitPosedge (a);
        wait(b&c,25):
    }
}
```

Listing 3.3: Example 2.

RT-SVA conversion for this property is as follows:

$$!a[\sim> 1]\ \#\#1\ a[\sim> 1]\ \#\#1\ !(b\ \&\ c)[*0:25]\ \#\#1\ (b\ \&\ c)$$

LPN translation for this example is shown in Fig. 3.9.

3. The delay between the second rising crossing of a at $2.5V$ and the first falling crossing of b at $4.5V$ is $250.0ns$ with a tolerance of $2.5ns$.

   This can be expressed as shown in the Listing 3.4 below.

```
property example3{
    real b;
    real a;
    always{
        assertUntil(b>45,a>=25);
        assertUntil(b>45,a<25);
        assertUntil(b>45,a>=25);
        assert(b>45,2475);
        wait(b<=45,50);
}   }
```

Listing 3.4: Example 3.

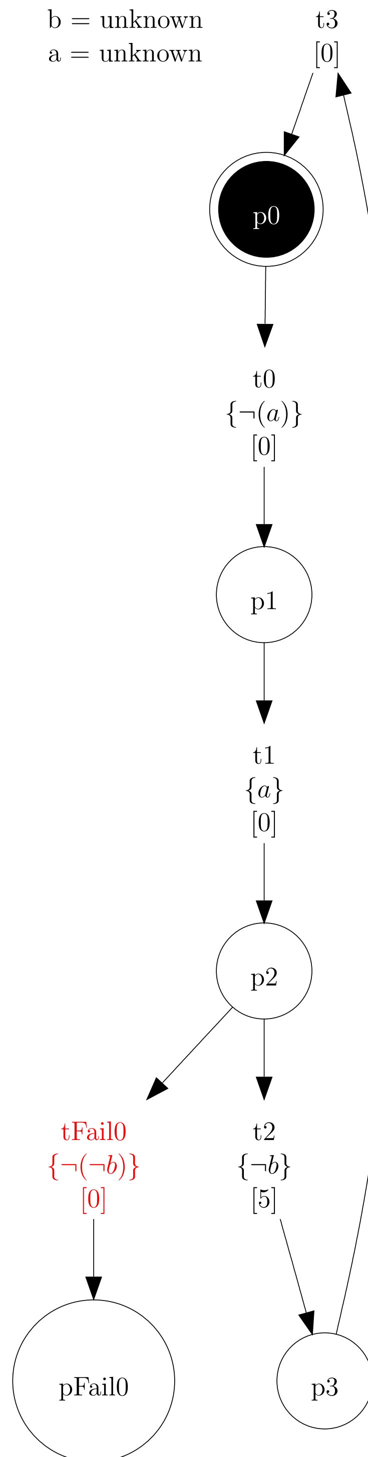


Figure 3.8: An LPN model for *example 1*.

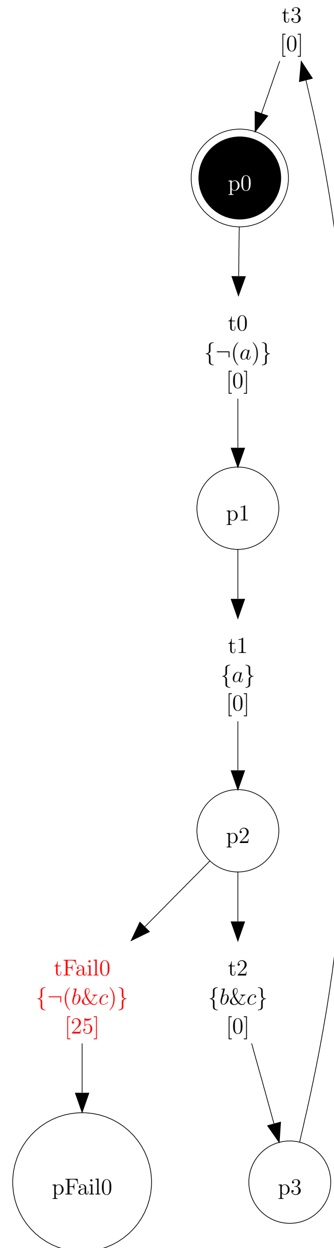


Figure 3.9: An LPN model for *example 2*.

RT-SVA conversion for this property is as follows:

$((b > 45)$ && $!(a \geq 25)[*0:\$]$ ##1 $(a \geq 25))$ ##1

$((b > 45)$ && $!(a < 25)[*0:\$]$ ##1 $(a < 25))$ ##1

$((b > 45)$ && $!(a \geq 25)[*0:\$]$ ##1 $(a \geq 25))$ ##1

$((b \leq 45)[*2475:2475])\ \#\#1$

$(!(b \leq 45)[*0:50]\ \#\#1\ (b \leq 45))$

LPN translation for this example is shown in Fig. 3.10.

4. Finally, phase interpoaltor property can be expressed as shown in the Listing 3.5 below.

```
property PhaseInterpolator{
    real ctl;
    real omega;
    boolean phi;
    always{
        wait(phi>=0);
        if(ctl=1){
                assert(omega<22,1375);
                wait(omega>=22,10);
        }
        else if(ctl=2){
                assert(omega<22,1315);
                wait(omega>=22,10);
        }
        else if(ctl=3){
                assert(omega<22,1255);
                wait(omega>=22,10);
        }
        wait(phi<0);
    }
}
```

Listing 3.5: Phase interpolator property.

RT-SVA conversion for this property is as follows:

$(phi \geq 0)[\sim> 1]\#\#1$

$\Big(((ctl == 1)\#\#0$

$((omega < 22)[*1375, 1375]\#\#1$

$!(omega \geq 22)[*0:10]\#\#1(omega \geq 22))\Big)$

or

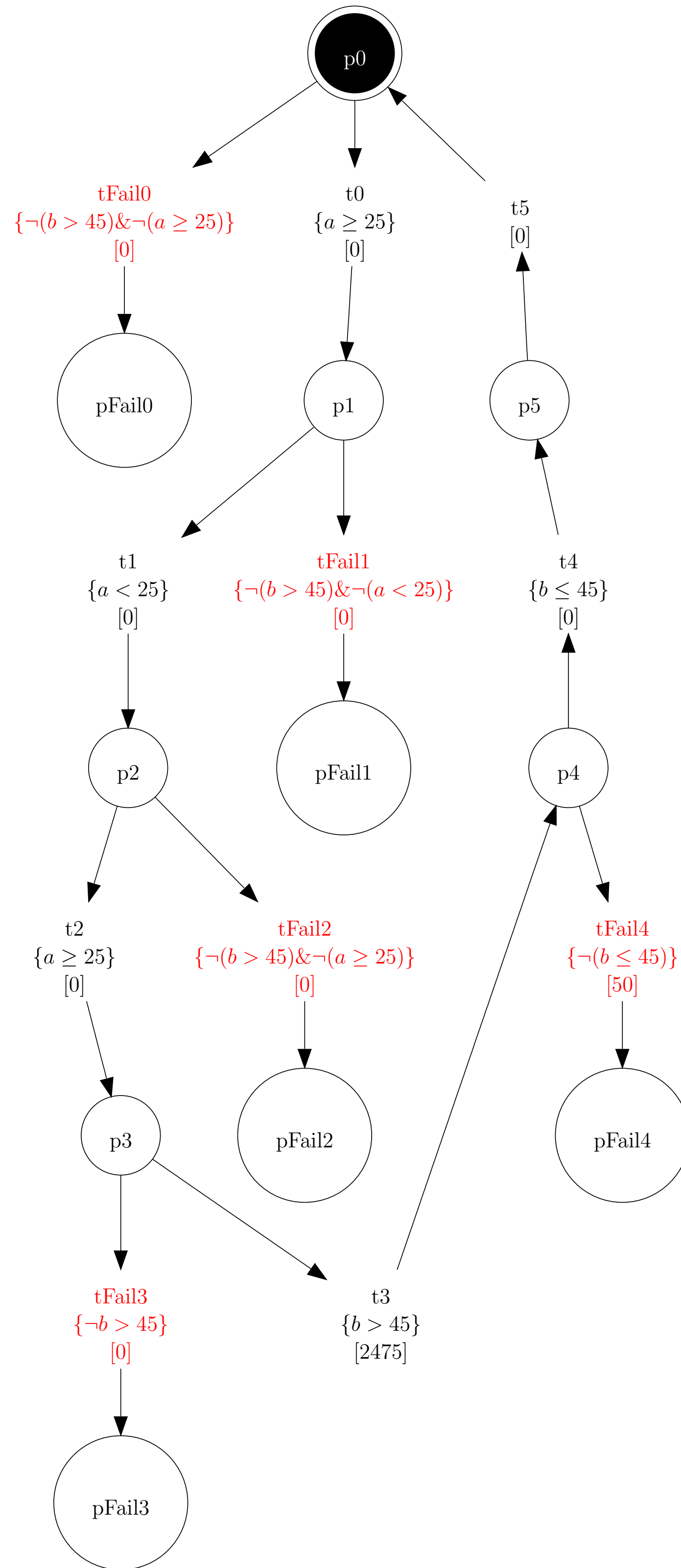


Figure 3.10: An LPN model for *example 3*.

$$\Big((ctl == 2)\&!(ctl == 1)\#\#0$$
$$((omega < 22)[*1315, 1315]\#\#1$$
$$!(omega \geq 22)[*0:10]\#\#1(omega \geq 22))\Big)$$

or

$$\Big((ctl == 3)\&!(ctl == 2)\&!(ctl == 1)\#\#0$$
$$((omega < 22)[*1255, 1255]\#\#1$$
$$!(omega \geq 22)[*0:10]\#\#1(omega \geq 22))\Big)\#\#1$$
$$(phi < 0)[\sim> 1]$$

Fig. 3.11 shows the LPN translation of this property.

3.6 Verification

LEMA is used to verify the phase interpolator circuit model with the property built using the property translator explained in the previous sections. First, LEMA is used to create LPN models for three different PI circuits that differed in the number of control inputs. These circuits allowed for 4, 8, or 16 different control values corresponding to 4, 8, or 16 different out phases, respectively. Fig. 3.12 shows the model generated for PI circuit with 4 control signals. Property to be verified against this model is shown in the Listing 3.6 below. Here, for *ctl* value of 10, property checks that output *omega* is low for 1699 time units and then goes high within the next 102 time units.

The results of the verification of the PI circuits for 4, 8, 16 inputs are given as the first three entries of Table 3.1. As can be seen from this table, the phase shift of the clock is successfully verified to be correct. To simulate an output clock that goes high too soon, the property is changed for the 4 control signals PI so that the assert statement for control value 40 asserts that *omega* is between 22 and 23 for 680 time units instead of 759. As is seen in the fourth entry of Table 3.1, the property correctly signals a failure. Each of these checks is done with an environment that could nondeterministically change the control signal shortly before the next time the input clock goes high. If this restriction is removed and the select signal is allowed to change at any time, a failure is detected, as indicated by the fifth result of Table 3.1.

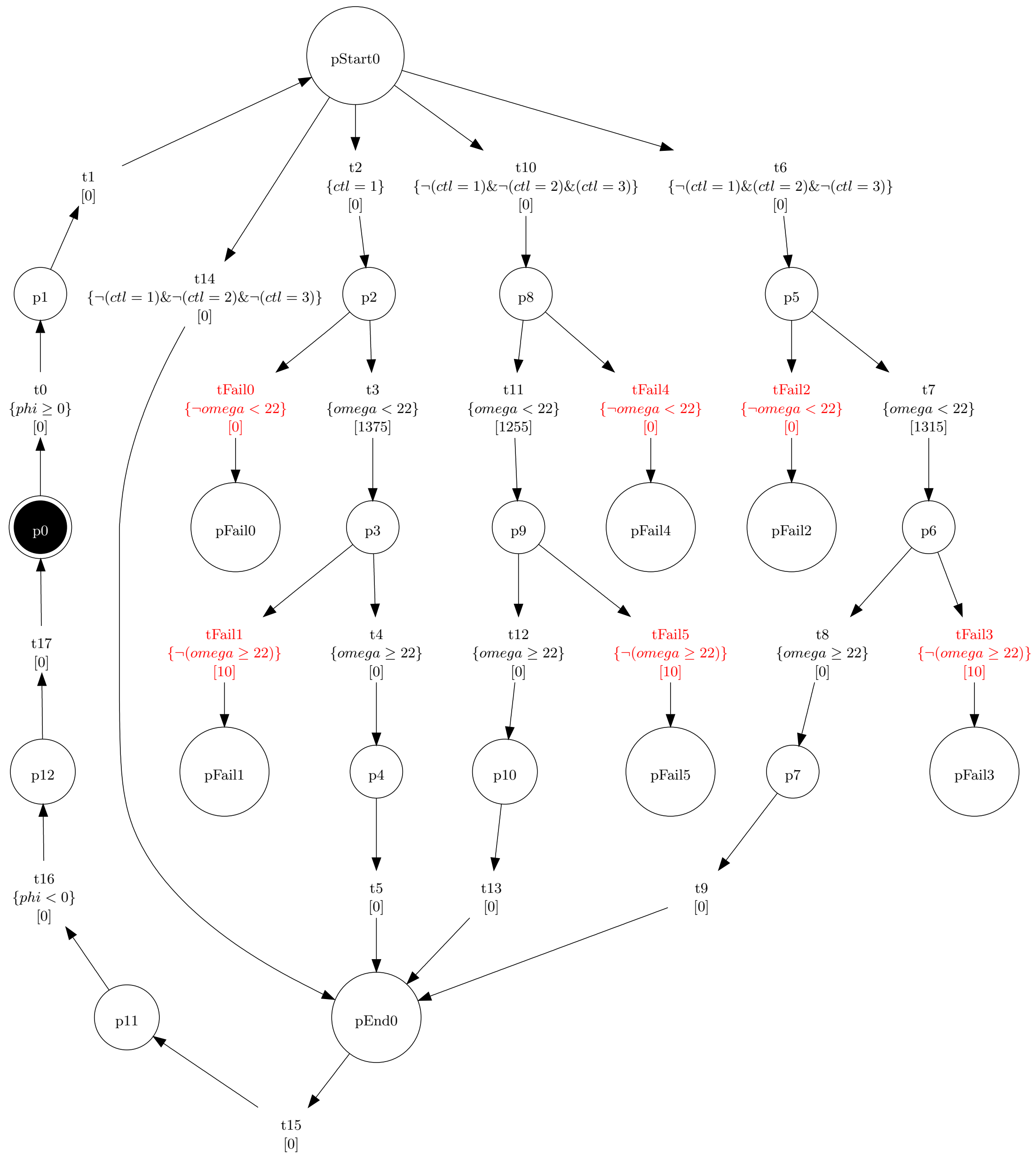


Figure 3.11: An LPN model for *Phase Interpolator Property*.

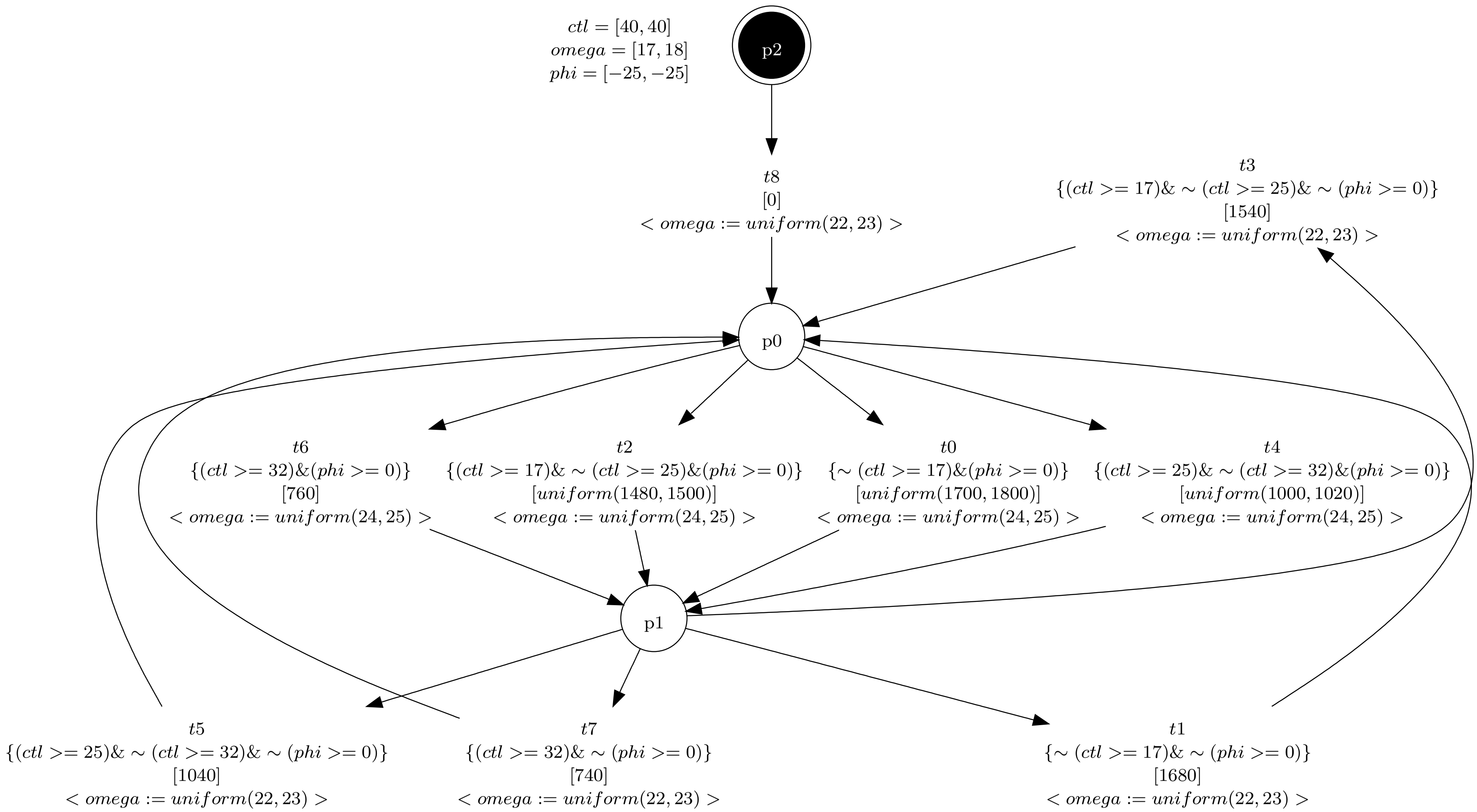


Figure 3.12: PI model to verify.

```
property PIVer{
          real phi;
          real omega;
          boolean ctl;
          always{
              wait(phi>=0);
              if(~(ctl>=17)){
                   assert(omega=uniform(22,23),1699);
                   wait(omega=uniform(24,25),102);
              }
              else if((ctl>=17) & ~(ctl>=25)){
                   assert(omega=uniform(22,23),1479);
                   wait(omega=uniform(24,25),22);
              }
              else if((ctl>=25) & ~(ctl>=32)){
                   assert(omega=uniform(22,23),999);
                   wait(omega=uniform(24,25),22);
              }
              else if(ctl>=32){
                   assert(omega=uniform(22,23),759);
                   wait(omega=uniform(24,25),2);
              }
              wait(~(phi>=0));
          }
}
```

Listing 3.6: PI verification property.

This failure occurs because after the property begins to check the output phase for one control signal, the environment can change the control signal to a different value, resulting in a different phase. The property then continues to check for the behavior for the previous control which results in a failure.

Table 3.1: Verifying Phase Interpolators.

Property	Time in s	States	Verifies?
PI With 4 Control Signals	0.135	126	Yes
PI With 8 Control Signals	0.277	300	Yes
PI With 16 Control Signals	1.362	769	Yes
PI With Short Delay	0.083	14	No
PI With Changing Controls	0.779	2407	No

CHAPTER 4

MODEL GENERATION AND INTERPOLATION

This chapter describes methods implemented for improved model generation and generalization of these models. Section 4.1 describes the motivation for these improvements. It shows the VCO model generated by LEMA and explains the limitations of this generated model. This section also explains how the current LPN generation methodology lacks the ability to separate the transient period from the steady-state of the circuit. Also, it states the need for generalization of these models. Section 4.2 and Section 4.3 describe the contributions of this thesis. In that, Section 4.2 discusses the steps taken to separate the transient state of the circuit from the steady-state with the addition of the *stable* variable. Section 4.3 focuses on generalization of the models by implementing a linear abstraction methodology, *interpolation*.

4.1 Motivation

The motivating example for this chapter is that of a *phase locked loop* (PLL). Fig. 4.1 shows a block diagram of a PLL circuit. A PLL is a control system that tries to generate an output signal whose phase is related to the phase of the input reference signal. The circuit consists of a variable frequency oscillator and a phase detector. This circuit compares the phase of the input signal with the phase of the signal derived from its output oscillator and adjusts the frequency of its oscillator to keep the phases matched. The signal from the phase detector is used to control the oscillator in the circuit. A typical PLL consists of the following circuit blocks.

1. *Phase frequency detector* (PFD) : This circuit has two inputs, one is the reference input and the other is the feedback from the VCO. It compares these

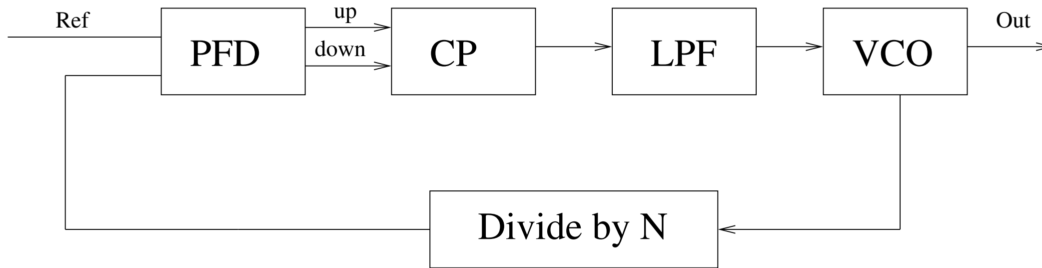


Figure 4.1: Block diagram for a phase locked loop.

two signals and outputs the 'up' or 'down' signal depending upon the phase difference between the two signals.

2. *Charge pump* (CP) : This circuit is a kind of DC to DC converter that uses capacitors as energy storage elements to create either a higher or lower voltage output. It gets input from the PFD and generates the voltage that is proportional to the phase difference between the two signals.

3. *Low pass filter* (LPF) : This circuit gets the input from the charge pump. The output from the charge pump is not smooth but has noise. The LPF is used to remove this noise. It is necessary for the stability of the circuit performance.

4. *Voltage controlled oscillator* (VCO) : This circuit is a variable frequency oscillator that generates the output whose frequency is set by the input control voltage.

5. *Frequency divider* : This is an optional circuit in the feedback loop. It is used to generate a wide range of output frequencies from the single reference input signal. This is particularly useful in radio applications.

PLLs are widely used in radios, telecommunications, computers, and other electronic applications. They may generate stable frequencies, recover a signal from a noisy communication channel, or distribute clock timing pulses in digital logic designs such as microprocessors. The notion is, frequency is a derivative of phase. So, keeping the input and output phase in lock step implies keeping the input and output frequencies in lock step. Thus, a phase-locked loop can track an input frequency, or it can generate a frequency that is a multiple of the input frequency. Based upon

the types of circuit blocks, a PLL circuit can be analog or digital. In an analog PLL, the phase detector is an analog multiplier circuit while in a digital PLL, it consists of digital circuits like XOR gates and edge-trigger JK flip-flops.

A block diagram of a digital PLL circuit is shown in Fig. 4.2. In this circuit, a *digital low pass filter* (DLF) and a *digital phase detector* (DPD) are the digital components. A *digital-to-analog converter* (DAC) and *time division converter* (TDC) are the AMS components while the VCO is analog in nature. The DPD detects the phase difference between the input wave and the reference wave. It is in digital format. To filter the noise, it is passed through the DLF. The digital output of the DLF is converted to analog with a DAC. It is then given as input to the VCO. The VCO output is given as input to the TDC. In the TDC, the 2 signals to be compared are passed through a chain of 64 delay elements. The difference in their phase at each stage is analyzed to convert the phase difference into a digital form. This is given as input to the DPD. We have considered the VCO circuit for our experiments.

As described above, any change on the input to the VCO produces a change in the frequency of the output signal of the VCO. Fig. 4.3 shows a VCO simulation trace which shows the VCO output for the control voltage of 2V. Fig. 4.4 shows the LPN model for the VCO circuit that is generated for control input values of 2V, 3V, and 4V. The simulation traces for these values are generated separately and then taken together to generate the model. In this model, after every control input change, the model immediately goes to the steady state and remains in that state until the next control input change. However, in AMS circuits, it takes time for the circuit to produce the steady-state output after the control input voltage is changed. In the VCO example, any change on input produces a change in the frequency of the output signal. After the change in input, the output does not produce the steady-state output frequency immediately. Instead, it goes through a transient period where the output frequency is not fixed. The transient causes the delay on transitions $t13, t23$, and $t28$ to have a wide variance which represents that the output frequency is never constant but keeps changing always. This does not correctly model the VCO circuit behavior. Thus, modeling this transient period becomes necessary to bring the model closer to its real behavior.

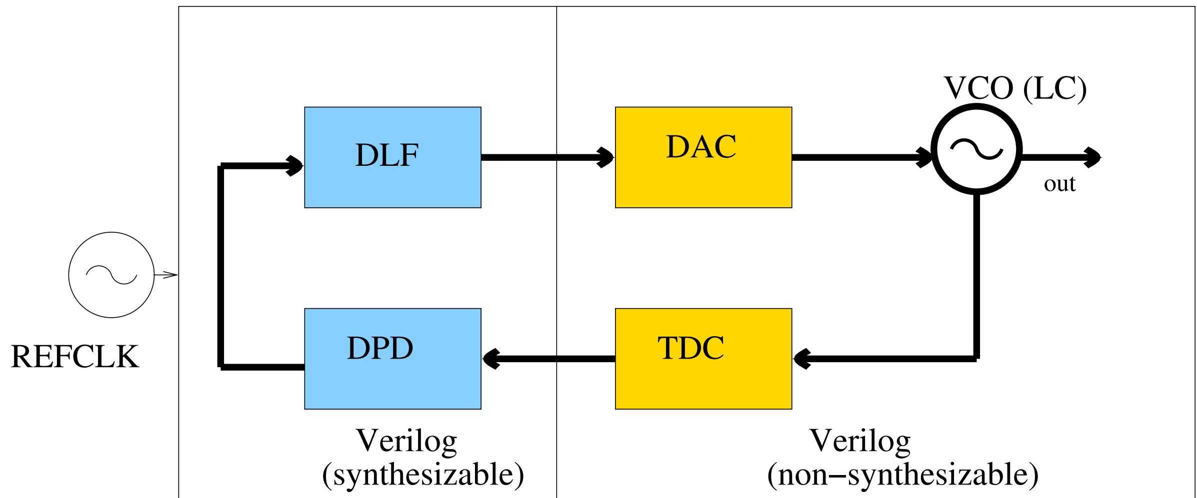


Figure 4.2: A digital PLL design.

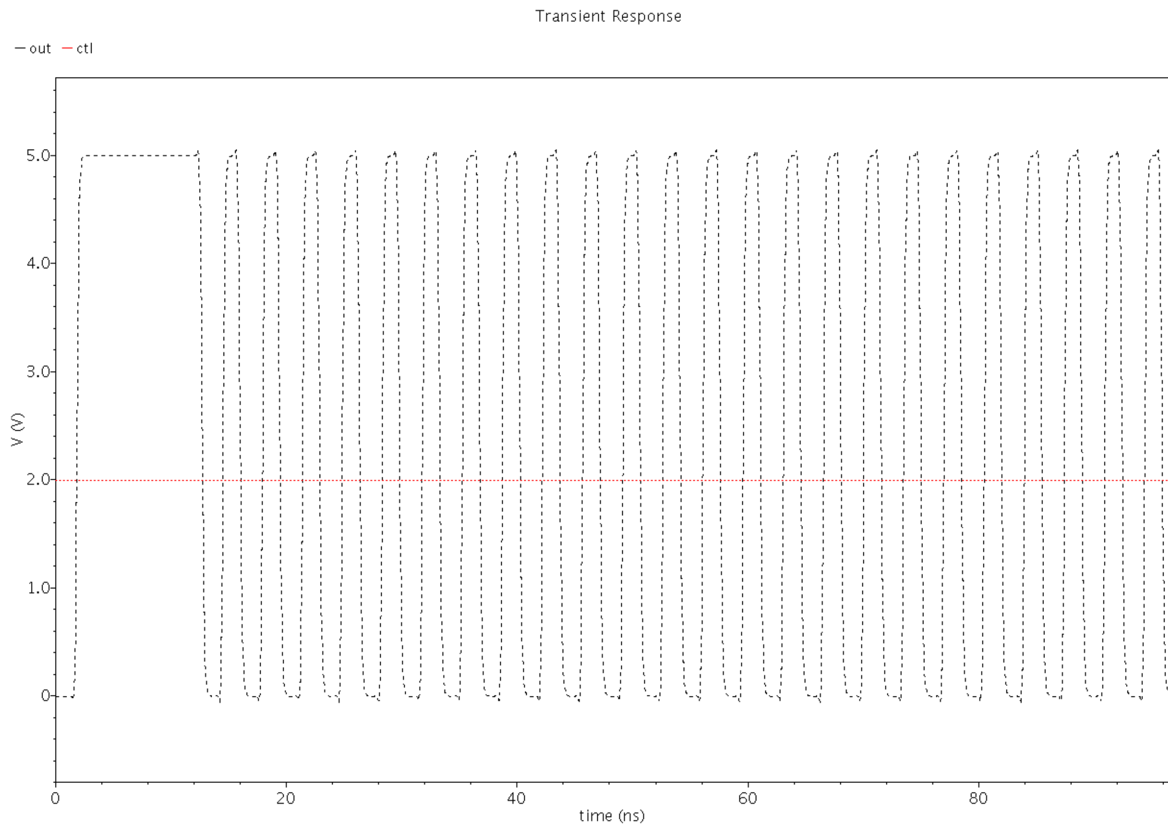


Figure 4.3: A simulation trace for a VCO.

As shown in Fig. 4.4, the model is generated only for the input values considered during the simulation. However, during verification, the circuit might get checked for the input values which are not considered during simulation. This calls for an exhaustive simulation of the circuit under test, but doing this is not often practical. For a VCO, there is an infinite number of possible control voltages, since it is a continuous signal. Therefore, instead of taking this approach, if the input-output relationship of the circuit signals is known, then models can be generated for the information not available directly through simulation data. We have implemented a linear abstraction, *interpolation*, to make the model more accurate.

4.2 Modeling Transient Behavior

In the LPN in Fig. 4.4, there are 3 loops which model the VCO behavior for 3 *ctl* input voltages, i.e., 2V, 3V, and 4V. As seen in the first loop, at transition $t20$, *ctl* input changes to 2V, and immediately (delay of 0) output is assigned a low value. After that, at transition $t24$ after the delay of 1.9 time units, the output is assigned a high value. After this, the model goes into a loop where the output is assigned a low and high value alternately modeled by transitions $t23$ and $t24$. As seen here, for $t23$, the delay is not fixed, but it keeps changing between 1.3 and 10.9 time units. This variance is a lot more than the actual time in the steady-state circuit behavior. Same for $t22$. This behavior is added in the model because of the initial transient period seen in Fig. 4.3. Thus, it is important to separate this transient state from the steady state. Improvement has been made in LEMA to capture this transient behavior. For this, a notion of a *control variable* is introduced. It is the variable, which, when it changes value, the output shows a transient behavior before settling down to its steady-state value. To capture this behavior, a new variable, *stable*, is added to the simulation data. The *stable* variable added to the data initially has a constant value of 1. Every time there is a change in the control input variable, the system goes through a transient period. The value of this *stable* variable is 0 during this transient period, and it is set to 1 when the system exhibits steady-state behavior. After setting the *stable* variable, the rates, values, and durations are again

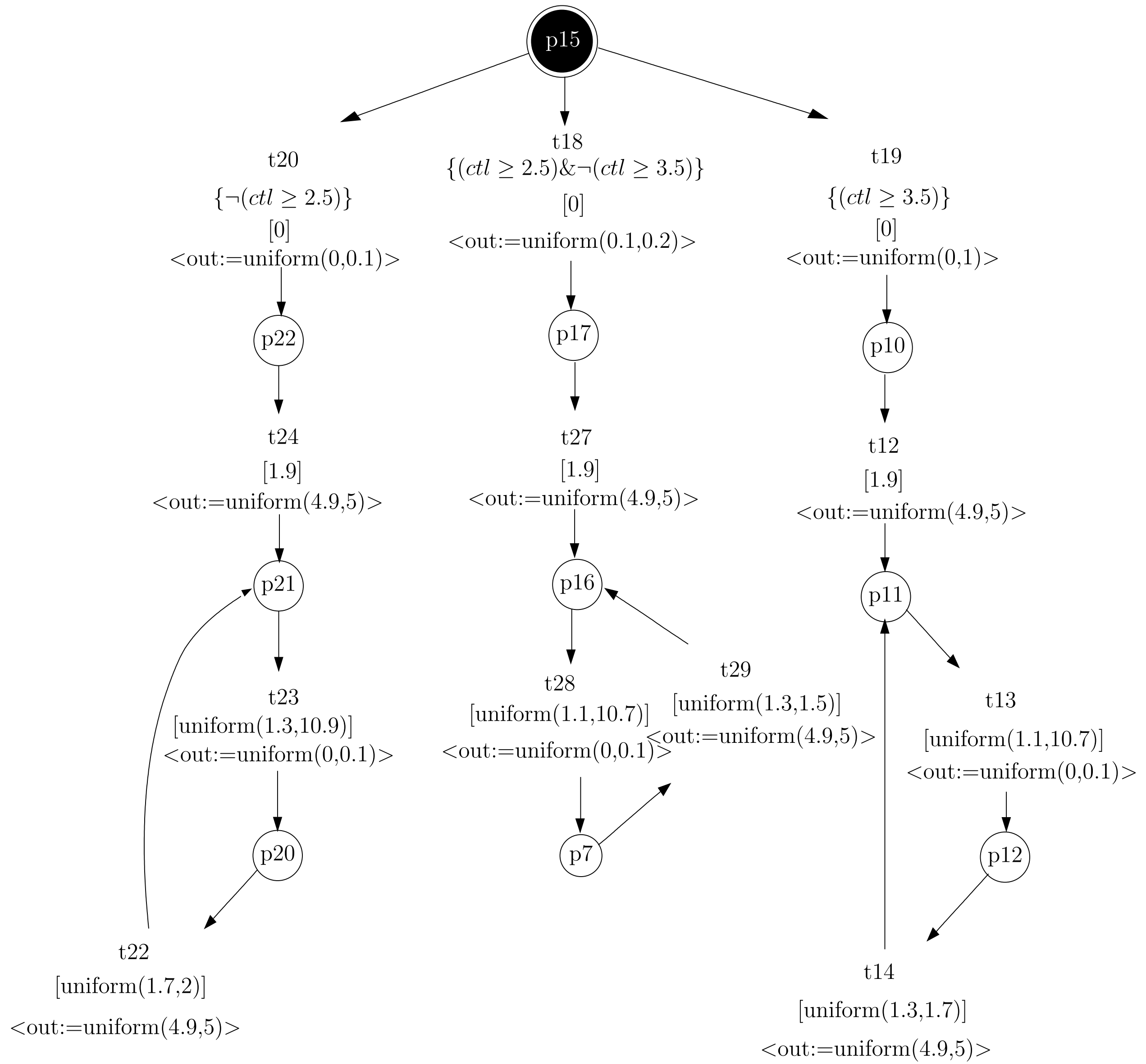


Figure 4.4: VCO LPN model.

calculated for the whole trace. The LPN is updated after this.

The *stable* variable is added to a simulation trace using the steps shown below:

1. Start at the beginning of the simulation trace ($i = 0$).
2. Search for the next change in a control input or the end of the trace, set this position to j.
3. Starting at $j - 1$, search backwards for changes in the output value, and at each change, record the duration it took for switching from one output value to another.
4. Check that these durations are the same as the previous duration within some tolerance.
5. If it is not within the tolerance, then mark this point k.
6. Mark all data points between i and $k - 1$ as unstable and from k to $j - 1$ as stable.
7. Set i to $j - 1$.
8. If i is not the end of the trace, go back to step 2.

The LPN for the generation of the *stable* variable is shown in Fig. 4.5. It sets the *stable* low initially, then it sets it back to high after the transient has passed. The *stable* variable is set low after every change on the control input. After the transient period has passed, it is set high. The model of the VCO after adding this *stable* variable is shown in Fig. 4.6. As seen in this model, at transition $t20$, the control input changes to 2V and the *stable* variable is low. Thus, it models the transient period of the circuit. $t23$ and $t22$ still have variable delay, but now it represent the transient period. After the *stable* has been assigned a high value, transition $t21$ fires, indicating that the system has gone into a steady state. $t25$ and $t26$ have fixed delays as is expected to be the case in the steady state.

This model can be further reduced by applying the functional approach discussed in Chapter 2. Here, the output is made a *care* variable. Thus, a new place is generated

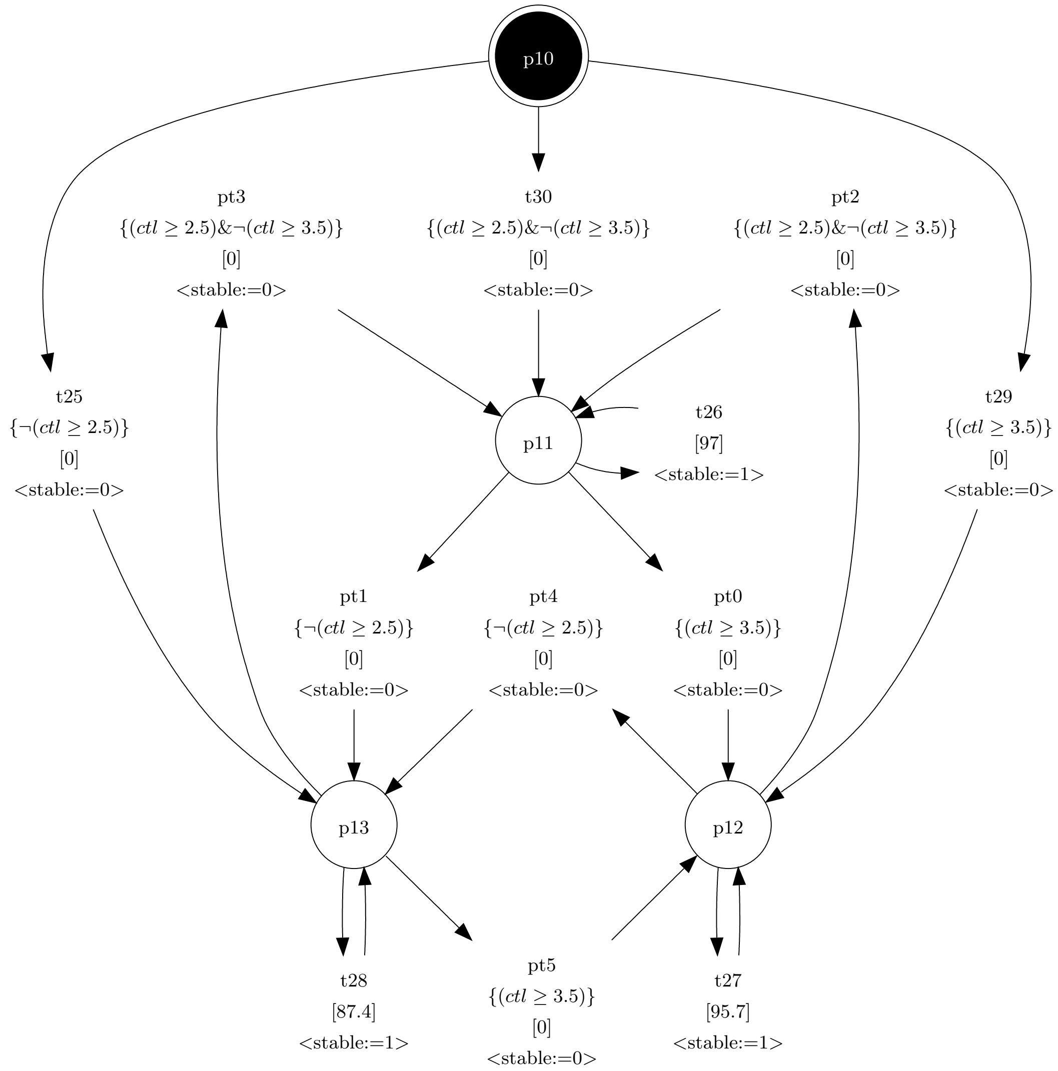


Figure 4.5: LPN model for *stable* generation.

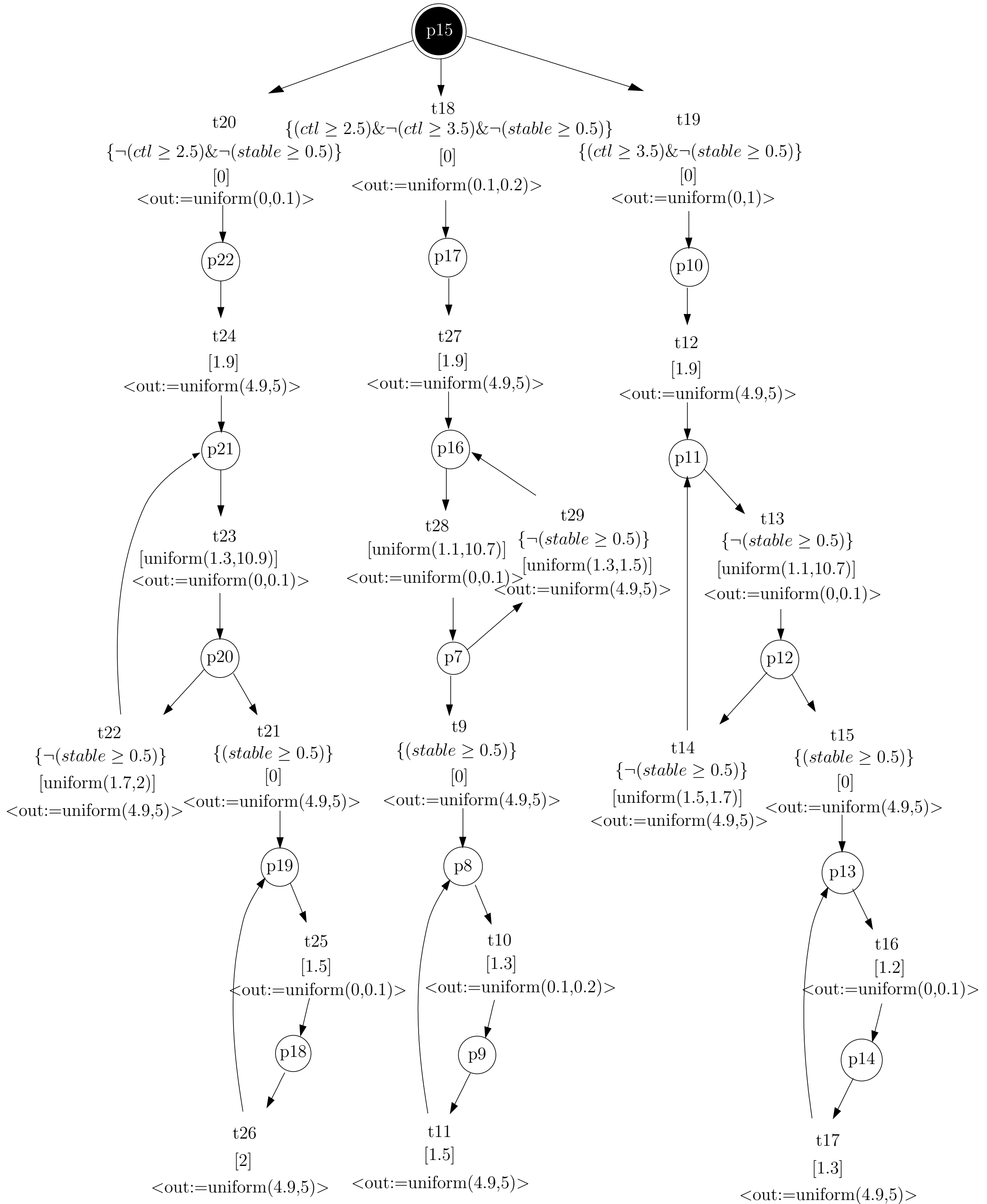


Figure 4.6: Generated LPN model for a VCO with a *stable* variable added to account for transient behavior due to changes in control input, *ctl*.

only when there is a change in the output value. The resulting model can be seen in Fig. 4.7. Here, at $t0$, *stable* is low, thus modeling transient behavior. When the output goes low, transition $t2$ is fired and when the output goes high, transition $t3$ is fired. The range of delays for these transitions are the functions of the control input values. Then, after *stable* goes high, transition $t4$ fires, taking the model to the steady state. When the output goes low, transition $t5$ is fired, and when the output goes high, transition $t6$ is fired. Delay values are fixed for the transitions and are functions of the control input values. In the figure, the transition delays are defined as follows:

f0 : $(\sim (ctl \geq 2.5)) * 1.9 + ((ctl \geq 2.5) \& \sim (ctl \geq 3.5)) * 1.9 + (ctl \geq 3.5) * 1.9$

f1 : $(\sim (stable \geq 0.5) \& \sim (ctl \geq 2.5)) * uniform(1.3, 10.9) + (\sim (stable \geq 0.5) \& \sim (ctl \geq 3.5)) * uniform(1.1, 10.7) + (\sim (stable \geq 0.5) \& (ctl \geq 3.5)) * uniform(1.1, 10.7)$

f2 : $(\sim (stable \geq 0.5) \& \sim (ctl \geq 2.5)) * uniform(1.7, 2) + (\sim (stable \geq 0.5) \& (ctl \geq 2.5) \& \sim (ctl \geq 3.5)) * uniform(1.5, 1.7) + (\sim (stable \geq 0.5) \& (ctl \geq 3.5)) * uniform(1.3, 1.5)$

f3 : $((stable \geq 0.5) \& \sim (ctl \geq 2.5)) * 1.5) + ((stable \geq 0.5) \& (ctl \geq 2.5) \& \sim (ctl \geq 3.5)) * 1.3 + ((stable \geq 0.5) \& (ctl \geq 3.5)) * 1.2$

f4 : $((stable \geq 0.5) \& \sim (ctl \geq 2.5)) * 2) + ((stable \geq 0.5) \& (ctl \geq 2.5) \& \sim (ctl \geq 3.5)) * 1.5 + ((stable \geq 0.5) \& (ctl \geq 3.5)) * 1.3$

4.3 Interpolation

In the previous sections, the LPN model for the VCO circuit is generated for the *ctl* input values in the simulation data. It is difficult and often impossible to simulate the circuit for all the possible input voltages. But sometimes, an engineer has to model the circuit for the data that are not available in simulation. To make this possible, linear abstraction methods can be used. We have implemented a type of linear abstraction, *interpolation*. This work assumes that though analog circuits are considered nonlinear, it is always possible to convert a circuit to a set of variables where the linear abstraction holds [36]. Applying this to the VCO circuit, we can safely assume that although the circuit is nonlinear, it can be converted to a domain

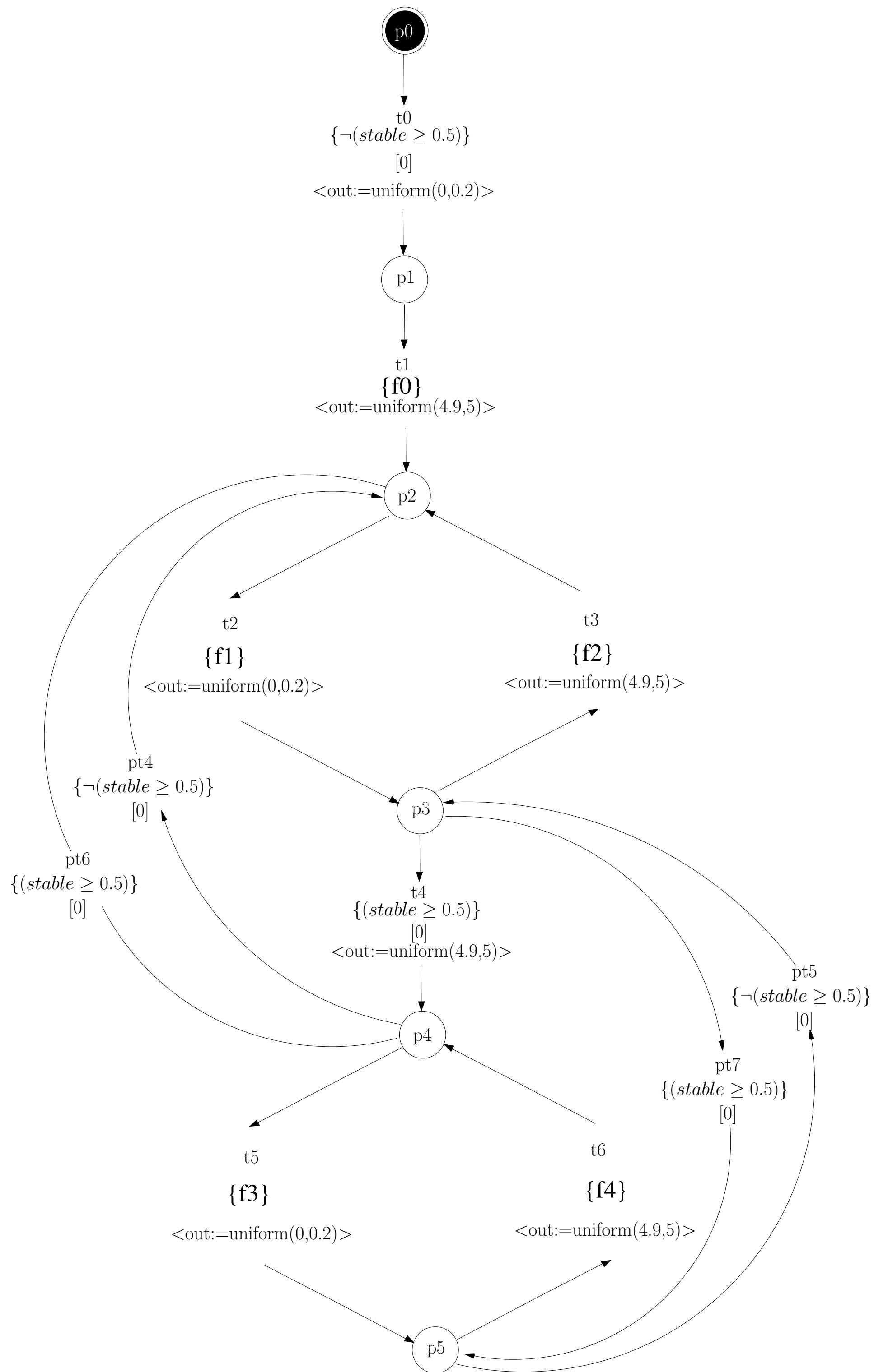


Figure 4.7: Functional VCO LPN model that accounts for transient behavior.

.

where its input and output are in a linear relationship.

To implement the interpolation, the following steps are followed:

1. For each transition t, consider its start region, end region, and the region of the control variable associated with that transition.

2. Then, find a transition t' where the start region and end region is the same but the region of the control variable has moved to the next highest region.

3. Get the delay associated with both the transitions.

4. Generate the delay expression using this data.

5. For every transition which has a control variable change associated with it, change the delay values with the expression generated above.

6. Change the enabling conditions on each of these transitions.

The straight line equation, $y = m * x + c$ is considered for interpolation, where,

$$\begin{aligned} y &= \text{'y' coordinate} \\ m &= \text{slope of the line} \\ x &= \text{'x' coordinate} \\ c &= \text{intercept of the line} \end{aligned}$$

In the VCO example, our method has to interpolate the output frequency generated for a particular control input value that is not present in the simulation data. In this case, the control input value is the $'x'$ coordinate. The delay on the transitions where the output changes as a response to the change in input gets translated to the output frequency of the signal. Thus, our method has to consider the delays on such transitions for interpolation. Thus, the *delay* is the $'y'$ coordinate. Starting from the lowest threshold, our method obtains pairs of $'x'$ and $'y'$ coordinates and obtains the delay expression for the region between two consecutive pairs, thus a *piecewise linear*

interpolation for the complete trace.

For the pair of coordinates, (x_i, y_i) and (x_{i+1}, y_{i+1}),

the slope $m = \frac{y_{i+1}-y_i}{x_{i+1}-x_i}$ and

the intercept $c = y_i - \frac{y_{i+1}-y_i}{x_{i+1}-x_i} * x_i$

The generated equation is,
$y = \frac{y_{i+1}-y_i}{x_{i+1}-x_i} * x + [y_i - \frac{y_{i+1}-y_i}{x_{i+1}-x_i} * x_i]$

After applying interpolation on the delay values, the delays in Fig. 4.6 get translated to the following form:

$\mathbf{f0} : (\sim (ctl \geq 3)) * 1.9 + ((ctl \geq 3)\& \sim (ctl \geq 4)) * 1.9 + (ctl \geq 4) * 1.9$

$\mathbf{f1} : (\sim (stable \geq 0.5)\& \sim (ctl \geq 3))*$
$\mathbf{uniform((ctl * (-0.2) + 1.7), (ctl * (-0.2) + 11.3))} + (\sim (stable \geq 0.5)\&(ctl \geq 3)\& \sim (ctl \geq 4))*$
$\mathbf{uniform((ctl * (0) + 1.1), (ctl * (0) + 10.7))} + (\sim (stable \geq 0.5)\&(ctl \geq 4))*$
$\mathbf{uniform((ctl * (0) + 1.1), (ctl * (0) + 10.7))}$

$\mathbf{f2} : (\sim (stable \geq 0.5)\& \sim (ctl \geq 3))*$
$\mathbf{uniform((ctl * (-0.2) + 2.1), (ctl * (-0.3) + 2.6))} + (\sim (stable \geq 0.5)\&(ctl \geq 3)\& \sim (ctl \geq 4))*$
$\mathbf{uniform((ctl * (-0.2) + 2.1), (ctl * (-0.2) + 2.3))} + (\sim (stable \geq 0.5)\&(ctl \geq 4))*$
$\mathbf{uniform((ctl * (-0.2) + 2.1), (ctl * (-0.2) + 2.3))}$

$\mathbf{f3} : ((stable \geq 0.5)\& \sim (ctl \geq 3))*(\mathbf{ctl * (-0.2) + 1.9}) + ((stable \geq 0.5)\&(ctl \geq 3)\& \sim (ctl \geq 4))*(\mathbf{ctl * (-0.1) + 1.6}) + ((stable \geq 0.5)\&(ctl \geq 4))*(\mathbf{ctl * (-0.1) + 1.6})$

$\mathbf{f4} : ((stable \geq 0.5)\& \sim (ctl \geq 3))*(\mathbf{ctl * (-0.5) + 3}) + ((stable \geq 0.5)\&(ctl \geq 3)\& \sim (ctl \geq 4))*(\mathbf{ctl * (-0.2) + 2.1}) + ((stable \geq 0.5)\&(ctl \geq 4))*(\mathbf{ctl * (-0.2) + 2.1})$

In the above equations, the new interpolated *delay* equations are highlighted.

This model generation takes around 1 to 2 seconds. Also, the difference in time taken to generate the model after addition of the *stable* generation and after the implementation of *interpolation* is roughly the same.

CHAPTER 5

CONCLUSIONS

Verification is becoming an important part of the overall design cycle with the increasing functional complexities of electronic systems. Simulation is a less effective approach towards validating the functionality of such large systems. Model checking methodology is becoming a widely used method to verify digital circuits. For this reason, it is very important to model the circuit at a level which is easy to build, use, and also preserves the behavior of the circuit under consideration. Boolean abstraction of digital signals has made modeling and model checking easier but the continuous nature of analog signals has made this task difficult. There is a need of a common and robust modeling methodology for AMS circuits.

5.1 Summary

For verifying the circuits, along with accurate models, it is important to have properties that can describe the required behavior. This thesis presents the methodology to specify temporal properties of the circuit and convert these to LPNs. This thesis also presents a methodology to improve the generated models. Major contributions of this thesis are as follows,

1. A new property language to specify the temporal properties of a system and its translation to LPNs.

2. Methodology to detect and model the unstable region of the circuit operation.

3. Generalization of the model by linear interpolation of the input signals.

5.2 Future Work

All these contributions have tried to address some problems on the modeling front. However, there is still a vast scope of research in this area. Listed below are some of the major areas for further exploration.

- Expanding the scope of property translator with more statements.
- Equivalence checking.
- Interpolation on the output values.

5.2.1 Expanding the Scope of the Property Language

The property language explained in the document is used to express properties of AMS circuits. The document also demonstrates some examples. These example properties are expressed using the statements that are explained in the document. To be able to specify more complex AMS properties, it is necessary to construct more statements such as a statement to check the circuit behavior where the transient state and steady-state are separated by addition of a *stable* variable.

5.2.2 Equivalence Checking

The model generation methodology can be used to check the equivalence between two circuits. It can be between an original circuit and its optimized version, or it can be two circuits designed in two different ways, for example, an analog PLL and a digital PLL. After generating the models for the two circuits with the same functionality, if the two models show the same abstract behavior, the circuits are considered equivalent.

5.2.3 Interpolation on the Output Values

This document has demonstrated the application of *interpolation* to generalize the models. But currently, this is being applied to the signals where the output parameter that gets impacted is *time*, for example: phase interpolator, where the output is the delayed version of the input. Thus, delay (time) is interpolated. However, in the

circuits like an amplifier, the parameter that gets impacted is the output voltage level. Interpolation will be a useful tool for analyzing such circuits.

REFERENCES

[1] M. H. Zaki, S. Tahar, and G. Bois, "Formal verification of analog and mixed signal designs : A survey," in *Design and Test Workshop (IDT), 2009 4th International*, 2009, pp. 1–1.

[2] A. Ghosh and R. Vemuri, "Formal verification of synthesized analog designs," in *International Conference on Computer Design*. IEEE Computer Society, 1999, pp. 40–45.

[3] K. Hanna, "Reasoning about real circuits," in *Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science*, vol. 859, 1994, pp. 235–253.

[4] ——, "Reasoning about analog-level implementations of digital systems," in *Formal Methods Syst. Design 16 (2)*, 2000, pp. 127–158.

[5] L. Hedrich and E. Barke, "A formal approach to verification of linear analog circuits with parameter tolerances," in *Design, Automation and Test in Europe*. IEEE Computer Society Press, 1998, pp. 649–654.

[6] ——, "A formal approach to nonlinear analog circuit verification," in *International Conference on Computer-Aided Design*, 1995, pp. 123–127.

[7] R. Kurshan and K. McMillan, "Analysis of digital circuits through symbolic reduction," in *IEEE Trans. Computer-Aided Design 10 (11)*, 1991, pp. 1350–1371.

[8] M. Greenstreet, "Verifying safety properties of differential equations," in *Computer Aided Verification Lecture Notes in Computer Science*, vol. 1102. Springer, Berlin, 1996, pp. 277–287.

[9] M. Greenstreet and I. Mitchell, "Integrating projections," in *Computer Aided Verification Lecture Notes in Computer Science*, vol. 1386. Springer, Berlin, 1998, pp. 159–174.

[10] ——, "Reachability analysis using polygonal projections," in *Computer Aided Verification Lecture Notes in Computer Science*, vol. 1569. Springer, Berlin, 1999, pp. 103–116.

[11] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards formal verification of analog designs," in *Design Automation Conference*. Association for Computing Machinery, 2004, pp. 210–217.

[12] T. Dang, A. Donze, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid system techniques," in *Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science*, vol. 3312. Springer, Berlin, 2004, pp. 14–17.

[13] M. Freibothe, J. Schonherr, and B. Straube, "Formal verification of the quasi-static behavior of mixed-signal circuits by property checking," in *Electron. Notes Theor. Comput. Sci. 153 (3)*, 2006, pp. 23–25.

[14] D. Nickovic and O. Maler, "Amt: A property-based monitoring tool for analog systems," in *Formal Modelling and Analysis of Timed Systems (FORMATS)*, 2007.

[15] O. Maler and D. Nickovi, "Monitoring temporal properties of continuous signals," in *Formal Modelling and Analysis of Timed Systems (FORMATS)*, vol. 3253. Springer-Verlag, 2004, pp. 152–166.

[16] T. R. Dastidar and P. P. Chakrabarti, "A verification system for transient response of analog circuits using model checking," in *VLSI Design (VLSID)*. IEEE Computer Society Press, 2005, pp. 195–200.

[17] A. Antoulas, D. Sorensen, and S. Gugercin, "A survey of model reduction methods for large-scale systems," *Contemporary Mathematics*, vol. 280, pp. 193–219, 2001.

[18] S. Gugercin and A. C. Antoulas, "A survey of model reduction by balanced truncation and some new results," *International Journal of Control*, vol. 77, no. 8, pp. 748–766, 2004.

[19] L. T. Pillage, X. Huang, and R. A. Rohrer, "Awesim: asymptotic waveform evaluation for timing analysis," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, ser. DAC '89. New York, NY, USA: ACM, 1989, pp. 634–637. [Online]. Available: http://doi.acm.org/10.1145/74382.74493

[20] N. Dong and J. Roychowdhury, "Piecewise polynomial nonlinear model reduction," in *Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2003, pp. 484–489.

[21] M. Rewienski and J. White, "A trajectory piecewise-linear approach to model order reduction and fast simulation of nonlinear circuits and micromachined devices," vol. 2, pp. 426–454, 2003.

[22] S. K. Tiwary and R. A. Rutenbar, "Scalable trajectory methods for on-demand analog macromodel extraction," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 403–408. [Online]. Available: http://doi.acm.org/10.1145/1065579.1065686

[23] P. Li and L. T. Pileggi, "Compact reduced-order modeling of weakly nonlinear analog and rf circuits," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 24, no. 2, pp. 184–203, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2004.837722

[24] R. Rutenbar, G. Gielen, and J. Roychowdhury, "Hierarchical modeling, optimization, and synthesis for system-level analog and rf designs," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 640 –669, march 2007.

[25] S. R. Little, “Efficient modeling and verification of analog/mixed-signal circuits using labeled hybrid petri nets,” Ph.D. dissertation, University of Utah, Dec. 2008.

[26] S. Batchu and C. Myers, “Automatic generation of abstract models from simulations of analog/mixed-signal circuits,” in *TECHCON 2010*, 2010.

[27] O. Maler, D. Nickovic, and A. Pnueli, “Pillars of computer science,” A. Avron, N. Dershowitz, and A. Rabinovich, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Checking temporal properties of discrete, timed and continuous behaviors, pp. 475–505. [Online]. Available: http://dl.acm.org/citation.cfm?id=1805839.1805865

[28] M. Boulé and Z. Zilic, “Automata-based assertion-checker synthesis of psl properties,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 1, pp. 4:1–4:21, Feb. 2008. [Online]. Available: http://doi.acm.org/10.1145/1297666.1297670

[29] K. D. Jones, V. Konrad, and D. Ničković, “Analog property checkers: a ddr2 case study,” *Form. Methods Syst. Des.*, vol. 36, no. 2, pp. 114–130, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1007/s10703-009-0085-x

[30] G. A. Sammane, M. H. Zaki, Z. J. Dong, and S. Tahar, “Towards assertion based verification of analog and mixed signal designs using psl,” in *Forum on specification and Design Languages, FDL 2007, September 18-20, 2007, Barcelona, Spain, Proceedings.* ECSI, 2007, pp. 293–298.

[31] S. Steinhorst and L. Hedrich, “A formal approach to complete state space-covering input stimuli generation for verification of analog systems,” in *Analog 2008: 10. ITG/GMM-Fachtagung Entwicklung von Analogschaltungen mit CAE-Methoden*, 2008.

[32] D. Smith, “Asynchronous behaviors meet their match with systemver- ilog assertions,” in *Design and Verification Conference (DVCON)*, 2010.

[33] O. Maler and D. Nickovic, “Monitoring temporal properties of continuous signals,” in *Formal Modelling and Analysis of Timed Systems (FORMATS)*, 2007.

[34] J. Havlicek and S. Little, “Realtime regular expressions for analog and mixed-signal assertions,” in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD ’11. Austin, TX: FMCAD Inc, 2011, pp. 155–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157654.2157679

[35] S. Mukherjee and P. Dasgupta, “Incorporating local variables in mixed-signal assertions,” in *TENCON 2009 - 2009 IEEE Region 10 Conference*, jan. 2009, pp. 1 –5.

[36] M. Horowitz, M. Jeeradit, F. Lau, S. Liao, B. Lim, and J. Mao, “Fortifying analog models with equivalence checking and coverage analysis,” in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010, pp. 425–430.

[37] D. Kulkarni, S. Batchu, and C. Myers, "Improved model generation of ams circuits for formal verification," in *TECHCON 2011*, 2011.

[38] D. Walter, S. Little, C. Myers, N. Seegmiller, and T. Yoneda, "Verification of analog/mixed-signal circuits using symbolic methods," in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 2008, pp. 2223–2235.

[39] S. R. Little, N. Seegmiller, D. Walter, C. J. Myers, and T. Yoneda, "Verification of analog/mixed-signal circuits using labeled hybrid petri nets," *Computer-Aided Design, International Conference on*, vol. 0, pp. 275–282, 2006.

[40] "Ams properties examples in http://www.eda.org/verilog-ams/htmlpages/public-docs/ams_assertions/amsproperties.pdf."