University of California
Santa Barbara

Automated Behavioural Identification and Timing Verification of Pulse Gate Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

David Donal Mc Carthy

March 2022

The Dissertation of David Donal Mc Carthy is approved.

Professor Luke Theogarajan

Professor Timothy Sherwood

Professor Tevfik Bultan

Professor Forrest Brewer, Committee Chair

December 2021

Automated Behavioural Identification and Timing Verification of Pulse Gate Systems

This thesis is dedicated to my family - my parents Michael and Elizabeth and my brother Tim.

Acknowledgements

Firstly I would like to thank Prof. Forrest Brewer for being my advisor through this research.

I would like to thank my parents Michael and Elizabeth and my brother Tim for their unwavering love and support both through this research and in my life generally.

I would like to thank my committee, Prof. Luke Theogarajan, Prof. Tevfik Bultan and Prof. Tim Sherwood for their insights.

I would like to thank Prof. Brewer, Prof. Philip Lubin and Prof. Theogarajan for supporting me as a graduate student researcher at various times throughout my thesis. Thanks also to the many professors with whom I have taught over the course of my PhD.

I would like to thank to the Electrical and Computer Engineering department for many teaching assistant opportunities throughout my studies, and also for the ECE dissertation fellowship. Thanks also to UCSB Graduate Division for the graduate division dissertation fellowship.

I would like to thank to my colleagues in the Brewer Lab, especially Merritt, Aditya, Prashansa and Carrie for their fruitful collaboration as we each worked on various aspects of pulse gates. Thanks also to my colleagues in the Lubin lab and Theogarajan projects for their collaborations on other research.

I would like to thank the friends I have made here at UCSB for their company and support throughout my time here. Thanks also to my friends in Ireland for their continued support.

Curriculum Vitæ
David Donal Mc Carthy

Education

2022	Ph.D. in Computer Engineering (Expected), University of California, Santa Barbara, United States.
2014	M.Eng.Sc. in Electrical and Electronic Engineering, University College Cork, Ireland.
2013	B.E in Electrical and Electronic Engineering, University College Cork, Ireland.

Publications This thesis is based on the following publications:

- Unit Time Modelling of Asynchronous and Pulse-Gate Circuits
  IWLS 2021, D. McCarthy and F. Brewer.
- High-Performance IP Design Using Pulse Logic
  Tutorial Presentation, VLSID 2021, F. Brewer, P. Mukim, D. Mc Carthy
- Automated Timing Constraint Generation for Pulse Gate Circuits
  IWLS 2019, D. McCarthy, M. Miller and F. Brewer
- Impolite High Speed Interfaces with Asynchronous Pulse Logic
  GLSVLSI 2018, M. Miller, C. Segal, D. McCarthy, A. Dalakoti, P. Mukim and F. Brewer

Other than this thesis, I have published:

- ROST-C: Reliability driven Optimisation and Synthesis Techniques for Combinational Circuits
  ICCD 2015, S. Grandhi, D. McCarthy, C. Spagnol, E. Popovici and S. Cotofana
- Predictable, Low-power Arithmetic Logic Unit for the 8051 Microcontroller using Asynchronous Logic
  MIEL 2014, D. McCarthy N. Zeinolabedini, J. Chen and E. Popovici

Additionaly I provided experimental assistance on the following related works:

- Design and Analysis of Collective Pulse Oscillators
  TVLSI 2019, P. Mukim, A. Dalakoti, D. McCarthy, C. Segal, M. Miller, J. Buckwalter and F. Brewer
- Distributed Pulse Rotary Traveling Wave VCO:Architecture and Design
  ISVLSI 2019, P. Mukim, A. Dalakoti, D. McCarthy, B. Pon, C. Segal, M. Miller, J. Buckwalter and F. Brewer

Abstract

Automated Behavioural Identification and Timing Verification of Pulse Gate Systems

by

David Donal Mc Carthy

This thesis addresses the problem of behavioural identification and timing verification for asynchronous, pulse-gate circuits. In particular, the design of very high performance logic and control functions such as time-to-digital, data-recovery, pipeline and FIFO control logic are targeted. The objective of this work is to produce models that are sufficiently general to include these hand designed circuits, amenable to automatic timing verification and, if possible, encompass known pragmatic techniques for asynchronous closure.

To achieve this timing verification a pair of timing models are proposed. The "unit time" model models the local behaviour of the circuit, and a "phrase" model models the communication between the unit timed regions. The phrase model also leads to a model for system level behaviour.

The "unit time" model allows the abstract system behaviour derived from symbolic simulation of the circuit as the specification. This step uses the special behaviour of pulse gates to create the symbolic abstraction, thus identifying possible nominal 'states' of model predicted behaviour and through this agreed behavioural convention the behaviour predicted by the model should agree with designer intended behaviour.

Next "phrases" are introduced as a model for communication between coherent unit timed regions, while the whole ensemble system need not be coherent. The whole system can be examined at a global level by modelling the set of phrases potentially currently occurring and which phrases result from these. This describes a path towards system level verification to ensure that behavioural escape does not occur due to overlaps of predicted local behaviours.

A 'coherence depth' property is defined to determine that gates in the local region are sufficiently connected over all logic cases for this model to be self sufficient. A formal proof that this property plus a set of critical path inequalities are sufficient to verify the timing of the system.

A computer tool is presented to derive the necessary timing constraints. Results from this tool are presented both to show the scale of problems it can handle, and that its constraints are in good agreement with SPICE for small circuits.

Contents

Chapter 1

Introduction

In this thesis, a two-part model is proposed for pulse-gate asynchronous circuit construction. This consists of a "unit time" local model and a "phrase" system-level/interface model. The objective in designing these models is to be:

- Sufficiently general to include hand-designed pulse gate circuits.
- Amenable to automatic timing verification.
- If possible, encompassing known pragmatic techniques for asynchronous closure.

Hand designed asynchronous circuits tend to be arbitrarily complicated at a local level. Nonetheless these designers include global symmetries that, if identified, can be exploited to allow verification. From a topological point of view, local components are strongly connected but global communication is much more organised.

Phrases are definitions of communications interfaces within the system. The phrases divide up the system into local region, within each of which the local behaviour is evaluated with the unit time model. Figure 1.1 shows this process. The phrases into and out each region become boundary specifications on it, with the inputs defining its behaviour

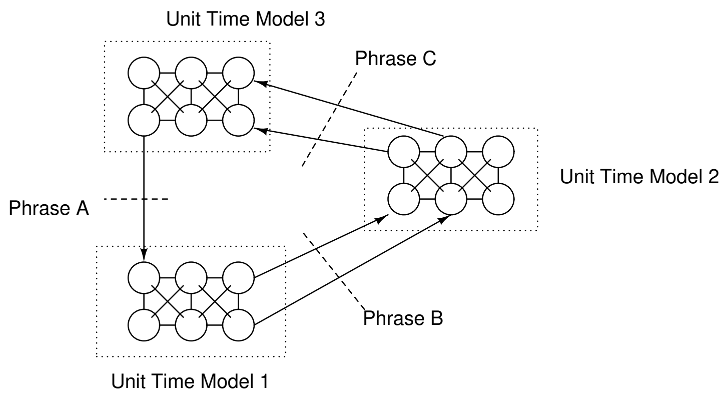


Figure 1.1: Example fragment of a pulse-gate circuit

and the outputs to be checked. Phrases also model system level communication and an outer view of phrases can be used for global level verification.

The unit time model is proposed as a local model designed to capture complex local behaviour. This model is based on the observation that designers must necessarily restrict the timing build-up within their local blocks, otherwise variance would build up indefinitely and the desired behaviour could not be guaranteed by any scheme. Thus, the unit time model makes strong timing assumptions, and together with the input specification from a phrase allows the nominal behaviours of the region to be inferred.

Timing verification in asynchronous circuits in general comes in two parts, verifying that the circuit is contained within some model so that the timing hazards predicted by the model are the only ones that can occur, and then verifying that those do not exist. Different models balance this in different ways, but all models need both, since the set of circuits that can work under completely arbitrary delay assumptions is behaviourally incomplete. Further, it may be necessary to make more timing assumptions to enable higher performance to be achieved.

Traditional approaches to asynchronous logic will be discussed in more detail later, but can be broadly divided into 3 categories. Asynchronous state machines as the name implies only consider local behaviours which can be descried as a state machine. Delay

Insensitive and Quasi Delay Insensitive designs require handshaking all the way down to the bottom level to eliminate as many timing assumptions as possible. Asynchronous pipeline models again consider only a very limited family of local behaviour, namely pipeline control elements and otherwise conventionally timed logic attached to them. The desire to capture a general variety of hand designed circuits which make timing assumptions freely for performance drove the need for a new model.

1.1 Pulse Gates

Pulse gates have signal propagation times that are 30-50% slower than conventional CMOS gates. Nonetheless, pulse gates represent a methodology to create circuits that operate at much higher rates than conventional CMOS logic in practice. This is due to execution with localized timing that is electrically co-incident with the data signal. Due to relatively high power density, high performance pulse circuits are not appropriate for generic logic functions, but are admirably suited for smaller, critical logic such as SERDES, link sub-circuits, FIFO/cache control and arbitration logic and selective clock technologies such as elastic pipelines.

Circuit designers using pulse gates have not used them arbitrarily but have used construction paradigms that allows exploitation of the high performance, but limits the timing complexity to predictable local timing arcs at the gate level. At larger scales conventional synchronisation strategies such as consensus, hand-shake and selective clocking are used. Thus, pulses can be gated or indeed subsumed by earlier pulses, but pulse arrival arbitration is handled by a separate circuit out of the timing model. These constraints act to limit the complexity of timing verification, avoiding the factorial complexity growth and logical incompleteness of general asynchronous circuits.

Several features of pulse gates motivated their design. They have relatively consistent propagation times. This means that circuit designers hand-designing circuits often make implicit timing assumptions, hence the need to perform timing verification. The same timing consistency also allows new timing models to be be explored. Pulse gate circuits also include some topological constructions such as local loops that are avoided in most (asynchronous) circuit construction strategies.

1.2 Unit Time Model

The unit time model is the local model of the circuit behaviour proposed. It is a topologically derived finite state machine model of an asynchronous circuit. The circuit is described as sequences of discrete “states”. A *state* is defined as the set of events present nominally simultaneously in the circuit, as well as the value of the data signals at that time. States are separated by the local sequencing of individual gates. The successors of a state are defined as the sets pulses present in the circuit simultaneously one topological, with latches updated if they have accepted pulses.

Given a circuit and an input specification, the unit time model through simulation (symbolic or otherwise) can identify a possible behaviour of the circuit, which is what the circuit would do if all gates in the circuit had the same constant nominal gate delay. In this unit delay view, the events occurring in the unit time model are a near-simultaneous group of pulses and the next state is the state of the circuit sampled one unit gate delay later.

By inspecting the behaviour of the circuit under the strong behavioural assumption, actions that need to occur in a certain order to produce that behaviour can be identified. That is the pairs of actions that constitute a hazard as defined in chapter 2 can be identified. Gate hazards are assumed to be a complete boundary on the space of event times for which a gate will behave correctly. Then, a looser assumption can be made that the circuit behaviour will be reached provided those key event pairs occur in the correct order, since for gate to behave differently would require a hazard to be violated.

The validity of this assumption, that the unit time model predicts an actual behaviour the circuit can be made to have, depends on the following factors:

- Timing dispersion that always exists does not become too large. A verifiable "coherent" property of circuits will be established to deal with this.
- No gate interaction produces different results between unit time model and real circuit. This is a restatement of the completeness of the hazard model ensuring correct behaviour.
- The unit time model is the only activity in the circuit or region where it is being applied.

The other issue is if the unit time model is applicable. That is, if the behaviour the unit time model predicts can actually be enforced on the circuit, and that it is the same behaviour the designer wanted. The assumption here is conservative. If the designer creates a correct circuit that violated the hazard model, it cannot be verified. However, a circuit that passes verification will exhibit behaviour compatible with the unit time model if it passes. The unit time model is particularly applicable to pulse gate circuits due to the typically narrow timing variance exhibited by a pulse gate set.

1.2.1 Coherence Depth

One of the key assumptions of the unit time model is that variance cannot build up indefinitely in the circuit. One way the amount of variance that can build up can be determined is to look at how long the pulses in the region of a system travel separately before all pulses remaining in that region can attribute their timing back to one common ancestor, or have timings as if they did. This is called the “coherence depth”. If the coherence depth is small then only a small amount of variance can occur. If it is larger then there is more opportunity for variance to build up, but this variance is still bounded by conservative estimation of gate properties. If the circuit is such that a common ancestor cannot be determined, then variations of event timing can build up indefinitely.

If timing dispersion is injected into the pulses in the same unit time state, then after the coherence depth states later, the circuit will have absorbed that dispersion and the relative timing between the pulses in that later state will be the same as if no dispersion had been injected since the pulses in the new state have picked either the pulse with the dispersion or the one without as their common ancestor. This property is the teleological value of the term ‘coherence’, it allows determination of when circuit behaviour allows direct comparison of timing of circuit events and thus enforcement of timing hazards of gates as well as completing the abstract notion of ‘state’ for the asynchronous circuit.

1.3 Phrases

The applicability of the unit time model is dependant on the coherence depth remaining small. Trying to build a unit time model of a large system will have an excessively large coherence and it would instead be better to build unit time models of different regions of the circuit and then model the communicating between those regions. Thus it is necessary to extend our unit time model to include communication. Since pulse gate circuits make timing assumptions across interfaces as readily as they do within themselves, it is necessary that coherence also be available across communication. On the other hand, where there are long gaps in communication, it does not make sense to preserve coherence and would be better to terminate that coherence reference and start with fresh coherence when the next communication occurs, e.g. in a serialiser system the timing between bits within a word might be within the same coherence but a new coherence models the timing of the bits of the next word.

Phrases are introduced as a system level model of communication. A “phrase” is a unit timed series of pulses and data-changes that represents one communication interaction between two regions of a circuit. A phrase consists of three parts:

- The “cut set”, a subset of the signals of the system.
- A regular language whose alphabet is logic cubes describing restrictions on the cut set signals.
- A phrase coherence depth.

The “cut set” is a set of signals which describes the boundary over which the communication is occurring. Using these cut sets, the circuit can be divided up into regions which can be unit time modelled independently. The phrase then describes the inputs into that region when executing the unit time model. The coherence depth for a phrase, like for the unit time model, is the largest number of steps needed to get back to a common ancestor between two events considered concurrent.

By considering which phrases produce each other, an outer system level model is theorised. This model allows the non-interfering property of pulse gates to be validated. This non-interference model means that a speed-independent view can be taken at the phrase system level.

1.4 Timing Proof

Since the objective of this work is to perform timing verification, it is first necessary to contain the amount of timing verification to be done. It is shown that for a unit time model with a finite coherence depth, it is sufficient to check single ancestor timing constraints up to a certain length, specifically where the shorter arm is of length of at most the coherence depth.

This is achieved by a three part induction building correspondence of both behaviour and timing bound between unit timed traces and real time traces. Given the current step exists behaviourally, applying the definition of coherence depth gives a general bound on the time between two pulses considered in the same state. This is then refined further by considering possible synchronising elements. This combined with the assumed timing verification then gives that the next step occurs as predicted.

1.5 Tool Implementation and Use

In this work, timing verification of the unit time model is achieved by deriving a complete database of path timing constraints that are checked post layout. This is in keeping with the methodologies of timing-based layout and design. In practice, the layout system creates an abstraction of the timing constraints to drive physical placement, followed by constraint checking. A design passing this is assured of correct operation assuming the timing assumptions are correct. It is also possible that a potentially correct design will be unable to be placed without timing faults – indicating the need for circuit or subsystem redesign.

1.5.1 Coherence Depth Evaluation

To measure the coherence depth, labels are added to a state in the unit time model representing whether an action is faster or slower than nominal, and symbolic simulation is performed to ensure these labels converge to the same at some future point. The number of steps taken is the coherence depth.

Since in the system level view, the coherence depth of a phrase depends on the region producing it, which in turn depends on the coherence depth of the input phrase of the region, the final answer must be found through iteration to a fixed point. The input coherence depth of each region is initially assumed to be 1, or can be user specified to be higher. Output coherence depths are measured similarly. If those output coherence depths feed into a region, and tell us the coherence depth of that phrase is higher than previously known we must retest that region to see if the increased input coherence depth increases the region or its outputs coherence depths. This is iterated until upper fixed point.

1.5.2 Hazard Tracing

The timing hazards that have to be verified against can be identified by inspection of the circuit schematic. To find these paths inequalities that lead to those hazards, a recursive backtracking search is performed over this to find possible ancestors of the hazard events and thus find paths. This search is topologically driven, and state sets are propagated throughout the search to ensure that states exist that lead to this trace.

Where a timing path in a region is traced back to the input boundary, this timing constraint on the inputs is back propagated into the region that produced that phrase and traced there as a new hazard. These are called 'through' hazards. Whether the correct signals occur is tested in the backtrace, and whether the output state agrees with the input state in the next is checked during the CTL recheck of the path.

1.6 Thesis Structure

- In chapter 1, a general introduction to this thesis has been given.

- In chapter 2 the background of this thesis will be discussed in two parts. One is related work generally, and one is an analysis of the behaviours and construction rules of pulse gate circuits
- In chapter 3 the “unit time model” is developed as a local model of pulse gate behaviour.
- In chapter 4 “phrases” are developed as a system level model.
- In chapter 5 the soundness of the unit time model is proved.
- In chapter 6 the construction of computer tools to perform the timing analysis of pulse gate circuits based on these models is discussed.
- In chapter 7 experimental results are presented
- Chapter 8 summarises the conclusions of this thesis.
- Appendix A describes the PySMV library constructed to facilitate this work.

1.7 Permissions and Attributions

1. Several pulse gate example circuits are form this work are from Merritt Miller’s Thesis[1]. These are used with permission and are cited where relevant. It is the circuit topology which has been reproduced, any drawings shown here are redrawn by me.

   Diagrams throughout this thesis were originally presented in my papers presented at the International Workshop on Logic and Synthesis, 2019 and 2021. For this workshop, copyright in the papers remains with the author so permissions are not an issue.

Table 2.1 is reproduced from a joint paper presented at the Great Lakes Symposium on VLSI 2018[2]. The conference proceedings for this are published by the Association for Computing Machinery (ACM), and this re-use is permitted by the ACM copyright policy, section 2.5 [3].

Chapter 2

Background

2.1 Related Work

2.1.1 Prior Pulse Gate Work

The pulse gate design style on which this work is based was initially introduced in [4]. It was first applied to high speed communications circuits [1]. It has also been applied to high performance oscillator designs [5, 6, 7]. It has also been considered for applications such as time-to-digital converters and multi-wire phase encoded links [8].

An alternative pulse gate timing algorithm appeared in [1]. This algorithm requires identification of frames where each gate only fires once, this technique did allow for looping behaviour (i.e. the circuit re-triggering itself), however, the manual placement of timing frames allowed for potentially missing possible behaviours. It was a manual methodology to organize the timing variables suitable for hand design. Further, that model is incapable of modelling pulse absorption, a behaviour used in stabilizing high performance clock phase generators.

2.1.2 Other Pulse Gate Design Styles

The notion of self-resetting gates can be traced to work by Sites and others at the dawn of NMOS technology[9]. The first systematic work was that of Martin and Nyström[10]. In that work, pulse gates were used as part of Quasi-Delay Insensitive design paradigm. One of the main issues in that work and other asynchronous work is where level based request-acknowledge handshakes are used, proper handling of the reset of the handshake signals adds both circuit complexity and potentially lowers system performance.

Various pulse gate based micropipelines have been proposed, including Sutherland and Fairbanks's GasP[11] and Greenstreet's Surfing Interconnect[12]. These circuits exhibited the relatively high performance potential of pulse-gate designs, but beyond the pipeline stage setup and hold issues, timing models were not developed.

Self Resetting CMOS is another pulse gate style aimed at data-path computation. Computation is performed with overlapping wide pulses, implying many timing constraints to ensure pulse overlap[13]. However, such circuits treated these gates as dynamic gates, locked in a governing synchronous paradigm. Thus while there are unique timing issues to this design style, the techniques used for timing static circuits can be adapted readily.

This use enabled high stage performance of clocked designs, notably Intel Pentium 4 arithmetic pipelines[14] (Intel used the term self-resetting domino for this work). Again, the governing synchronous clock (at 1/2 the state rate for Intel) cast the timing problem back into the synchronous model.

2.1.3 Finite State Machine Model Checking

Finite state machines (and similarly, finite automata) [15] are useful models for systems. The equivalence with regular languages allows switching easily between a state based and trace based view, and semantics for non-determinism are well established. A large number of verification schemes exists for finite state machines. Symbolic Model Checking[16] encodes sets of states and the transition relation of a state machine as boolean functions, and thus allow checking of properties on sets of states of the state machine simultaneously by logic operations. Existential Quantification is required, thus Binary Decision Diagrams (BDDs)[17] are usually used as the logic representation since existential abstraction is relatively efficient in BDDs. However use of BDDs for large problems is limited by the fact that they are only efficient if a good variable order can be found, and a good variable order may not always exists[18].

Bounded Model Checking[19] in its original form, unwinds many copies the transition relation forward from the initial step to show that a violation cannot occur within a certain number of steps. k-Induction[20] combines this with a second check unwinding back from the set of property-violtating states, and thus allows the property to be proved for all time. More modern approaches such as Property Directed Reachability[21] use arrangements other than direct unrolling to achieve the same result more efficiently.

When model checking state machine models, the property to be checked can either be an invariant on a checking signal built into the machine that must be always true or false, or it can be a more general specification about state progression. Linear Temporal Logic (LTL) and Computational Tree Logic (CTL)[22] are two logics commonly used to specify such requirements. LTL specified requirements on traces produced by the state machine (thus taking a language view), and allows specifications such requiring as a property to be true eventually, always or in the next step of the trace. CTL imposes requirements on properties from a starting state of an state machine, such as a property being eventually true on all paths out of that state, or being always true for some path out of the state.

Symbolic Model Verifier (SMV)[16, 23] is the original state machine verifier, performing BDD based symbolic model checking with CTL specifications. NuSMV[24] is a reimplementation that additionally supports bounded model checking. Additionally the SMV input syntax is widely supported by newer solvers such as NuXMV[25] or eBMC[26] due to its use for expressing model checking contest problems. For these reasons and to allow the use of external tools, SMV syntax is used in this work.

The BDD operations underlying verification are widely implemented using the CUDD (Colorado University Decision Diagrams) package[27]. For SAT solving the MiniSat[28] solver is commonly used in automata tools (inlcuding NuSMV and property-directed reachability) due to its stable programming interface. More recently PicoSAT[29] is also used similarly. SAT solvers are an active area of research in themselves, yielding newer solvers such as CaDiCaL[30] and the Maple family of SAT solvers[31]. These focus on solving the largest possible single SAT instances and thus can have large overheads on the many small-to-medium SAT instances that are repeatedly checked in model checking problems, so MiniSAT and PicoSAT remain popular there.

2.1.4 Asynchronous Timing Verification

Asynchronous State Machines

Huffman's Fundamental Mode circuits[32] took an asynchronous state machine approach, and consisted of circuits where signals must arrive either close together and be considered part of the same transition, or be separated enough to arrive after that transition has settled. Building such state machines required extensive hazard removal.

Burst Mode logic focused on enabling the design of such machines in a hazard free fashion. This is achieved by limiting the communications between them to specified sets of signals ("bursts") all of which must change before the state machine transitions. It was originally proposed by Coates, Davis and Stevens[33].

Nowick showed that two-level hazard free implementation of burst mode and extended burst mode automata is possible by careful state assignment [34].

Much work has been carried out on synthesising burst mode and similar asynchronous state machine styles from various specifications is possible, often using Petri net style specific. In particular Chu[35] introduced the Signal Transition Graph specification style. Wilcox [36] presented another synthesis approach using the same specification. Cortadella's Petrify[37] more explicitly embraces the Petri net nature of the specification.

Relative timing[38] is a method of identifying timing inequalities in extended burst-mode circuits, to allow the optimisation of slower hazard free structures into faster ones with hazards that can be checked. The overall approach is somewhat similar to this paper, in that intended behaviour can be read from circuit structure. The local composition rules used in that work imply that each signal is only used once and so inequalities can be written in terms of the occurrence times of signals. In pulse gate circuits, designers frequently include looping behaviour even on a local level so it is necessary to analyse circuits where gates are used multiple times per analysis scope, leading to inequalities based on paths rather than individual signals.

Speed Independent and Delay Circuits

Speed Independent[39] circuits are circuits whose behaviour remains equivalent under arbitrary assignments of gate delays. Muller introduced the notion of speed independence, and a lattice definition for behaviour equivalence. Muller also introduced the property of "Semi-Modularity", which is where the system does not allow gates that have been excited (wanting to change their output) to become un-excited by any means other than firing. Muller showed that this semi-modularity property is sufficient but not necessary to give speed independence.

Delay Insensitivity[40] is the further requirement that the circuit be sound under any assignment of wire delays. This was originally proposed by Molnar and Clarke in their "Macromodules" project. They proposed the construction of a small library of pre-designed small modules, which could then be assembled into systems in delay insensitive systems. Many different module sets have been considered in subsequent works, Patra and Fussell[41] present one such set that is a minimal number of distance modules while retaining behavioural completeness.

Martin showed that if a circuit is required to be delay-insensitive[42] down to individual gates, then such circuits could only compute a very restricted set of functions. This led to Quasi-Delay Insensitive (QDI)[43, 44] logic which was delay insensitive apart from specific nodes where the wire delay is assumed to be matched. Applications of this focused on compilation in a correct-by-construction fashion. The specifications used are based on a hardware version of Hoare's Communicating Sequential Processes[45] model.

Beerel's Proteus[46] is a logic synthesis methodology for dynamic domino pipelines, which intentionally sizes the domino gates to have a unit delay. This allows for timing optimisation at a few-gate level but larger timing synchronisation is still achieved in a Quasi-delay-insensitive manner

Asynchronous Pipeline Styles

Building the entire circuit in a delay insensitive fashion has a large overhead. A variety of asynchronous pipeline control stages have been proposed, whereby the data computation is a relatively conventional pipelined static logic pipeline, with the latches being clocked by asynchronous handshakes. This was originally proposed by Sutherland in his "Micropipelines" model [47].

In Sutherland's original formulation of micropipelines, most of the logic complexity was spent making dual-edge sensitive control structures to avoid the reset side of the handshake. Thus various pulse versions have been considered to alleviate this, as has been discussed above.

Symbolic Time Verification

Belluomini[48] proposed an approach using numerical decision diagrams to represent explicit values of event time, and a using a partially ordered set (POSET) to constrain the timing problem.

Timed automata[49] are an extension to the standard automata model to allow verification of systems with time constraints. Real times are added to the input language alongside the input symbols. The state of the system includes real time clocks in addition to the usual state, which can be reset to zero by a transition being taken, and then count up real time. Transition guards in the automata can include comparisons between these clocks and a constant. UPPAAL is a verifier for such systems[50].

Maler[51] applied timed automata to verifying asynchronous circuits.

2.1.5 Other Relevant Work

One feature of pulse logic which will be discussed later is the extensive use of combinational loops within circuits. In conventional synchronous logic the “local” evaluation, the static gates between the registers must be a directed acyclic graph. As has been seen above it is also relatively rare in asynchronous logic. One work that did deal with this is Riedel’s Cyclic Combinational Circuits[52]. Compared to usual assumption that combinational circuits are acyclic, Riedel’s model allows circuits which have cycles in their topology provided that the evaluation of the circuit is acyclic since any cycle must be cut by the data values, e.g. muxes selecting inputs that do not form a cycle. In a correct Riedel circuit, the order in which gates evaluate is allowed to be data dependent, for each data assignment an evaluation order can be found with each gate evaluate to its final value at most once.

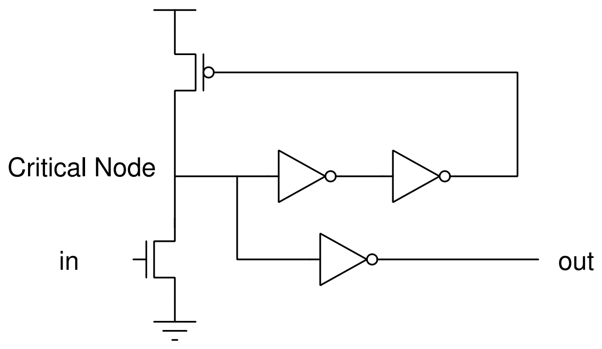


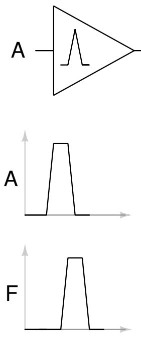
Figure 2.1: Pulse gate implementation

2.2 Pulse Gates

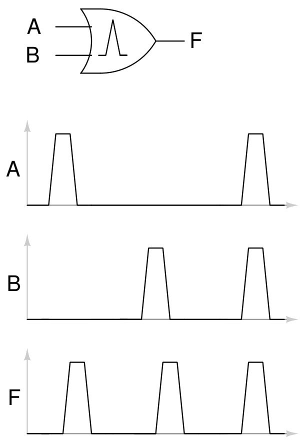
In this section, pulse gate circuits as they are designed by human designers will be discussed. Construction rules for pulse gate circuits will be discussed, as will the nature of timing hazards in pulse gate circuits.

A pulse gate circuit consists of two types of gates, the pulse gates themselves and pulse-actuated SR latches. A pulse gate is a self resetting CMOS gate. Figure 2.1 shows the most basic example, a pulse buffer. An arriving pulse pulls down the critical node. The output then starts rising. After the propagation delay of the reset loop, the critical node is pulled back up by the reset transistor, and the output goes low to end the pulse. The gate is sized so as the pulse is “round topped”, ensuring that rising and falling times are independent of input excitation[4]. Thus the pulse shape is largely dependent only on the arrival time of the actuating pulse, and not slope or amplitude.

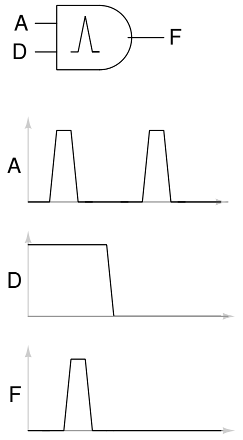
In addition to the basic pulse buffer shown, logic in pulse gates is implemented in two basic ways. One is by ANDing a pulses with a static level from a SR latch, or a more involved guard combining multiple latch values, producing an output pulse only if the latches have a required value when at the time the input pulses arrive. The other is by ORing two different arriving pulses together, producing an output pulse when either input pulse arrives. Both of these are achieved by replacing the *in* MOSFET with a



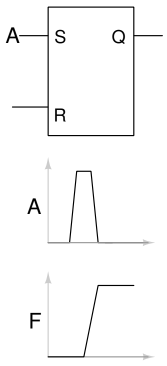
(a) Pulse Buffer



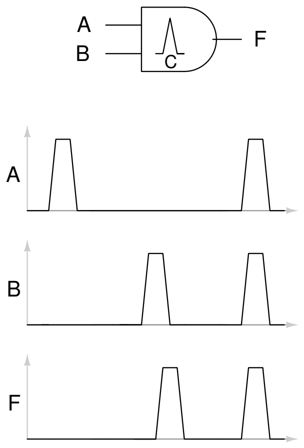
(b) Pulse OR Gate



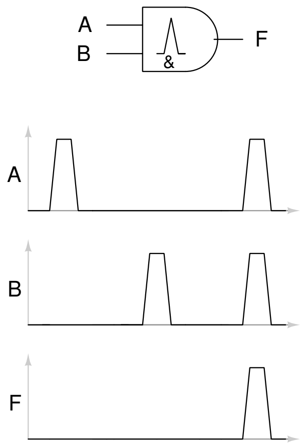
(c) Pulse AND Gate



(d) SR Latch



(e) Pulse Consensus Gate



(f) Pulse Conjunction Gate

Figure 2.2: Pulse gate logic elements

either a series or parallel NMOS pull down network. More general gates (e.g. 22 And Or) can also be constructed in this fashion.

Latches are typically CVSL-style SR latches sized to be correctly actuated by a pulse. Earle latches with logic in the latch pull down network can also be built. Figure 2.2 shows the basic pulse gate logical elements.

Several more elaborate pulse gate behaviours can be constructed for synchronisation. A pulse consensus gate has two input pulses, and it only produces an output pulse after both inputs have been activated. If only one is activated at a given time, it retains this in internal state, and when a pulse arrives on the other input the output is then produced. This is effectively a pulse equivalent to a Muller C element. A pulse conjunction gate is a gate that only produces an output pulse when it receives two simultaneous or near simultaneous pulse inputs.

When a pulse-OR gate is activated by two incoming pulses sufficiently nearby in time, it will cleanly produce one output pulse. This behaviour is referred to as “Coalescence” when it is designed. The timing of this pulse is based off the first arriving input pulse. It can be used in a synchronising element where the margins can be ensured.

2.2.1 System Construction

A typical example of a pulse gate circuit is the binary counter shown in Figure 2.3 (from [1]). It takes an input pulse train, and uses Pulse And and Or gates to filter the incoming pulse into pulse signals encoding both timing and state information that are used to update latches creating the effective next-state data signals. There are many possible ways to construct such a counter, with a trade-off between performance and complexity. At one extreme, the circuit consists of a single pulse being routed between pulse based toggle latches updating bits serially. At the other extreme multiple pulses

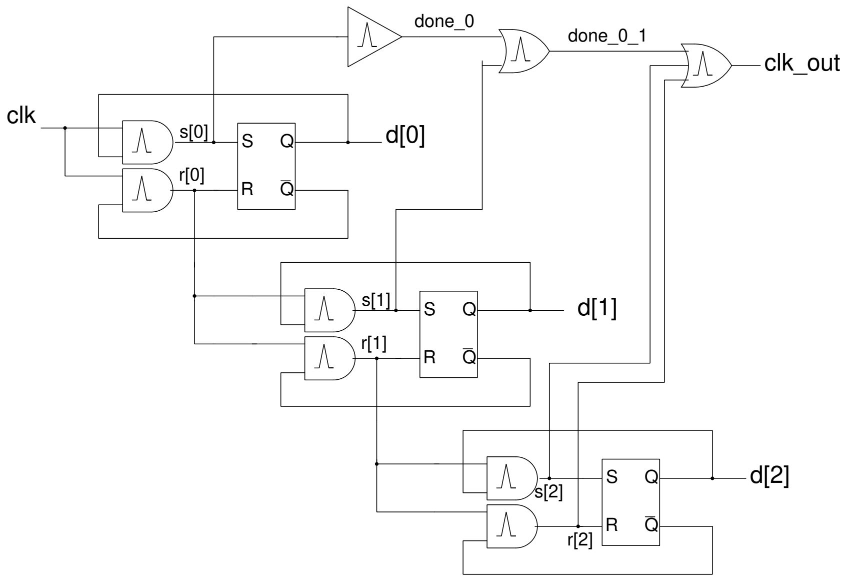


Figure 2.3: Pulse Gate Binary Counter

are selectively produced and simultaneously update several bits in race in consensus mode. The ability to readily produce data-laden signalling events which are locally timed accounts for the potential high performance, and these behaviours must be addressed by a practical verification strategy. Most commonly on a local level a 1-of-N code is used but at a system level other codes, such as a pulse order code, have been investigated [8].

The construction rules for pulse gate circuits are:

- Pulse gate circuits are strictly typed so that timing information is strictly derived from pulse gates, and not from latch outputs which are only used for guards.
- Two pulses arriving at a pulse gate are far enough apart to cause two completely separate firings of the gate, unless intentionally designed to coalesce.
- Two pulses arriving at an Pulse SR latch do not arrive so close as to cause meta stability or an indeterminate result.
- Data signals are stable when being sampled by a pulse.

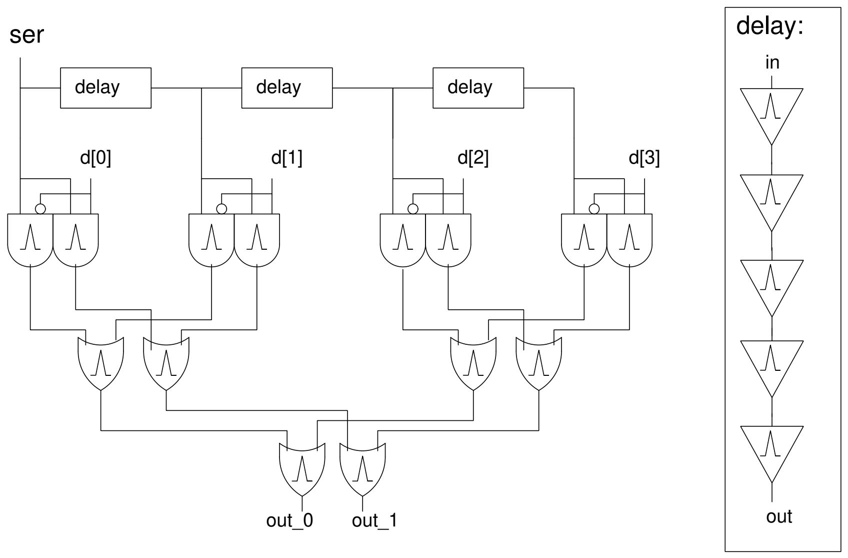


Figure 2.4: Pulse Gate 4-bit Serialiser

Several strategies for local coherence are used in pulse gate circuits, one is depositing data in latches for another pulse to read (the pulse timing path going into the latch terminating and timing then being defined by the other pulse path). The other is using coalescence to pick the first of two pulses arriving at an OR gate. While this can only accommodate a small amount of dispersion (before gate electrical rules are violated) it can easily prevent these small amounts of dispersion from accumulating over time. At the system level, pulse consensus gates are the most common since they can accommodate arbitrarily large amounts of dispersion. Of course they can also be used at locally.

Many of the standard asynchronous design styles can be constructed with pulse gates within these rules. However, pulse gates circuits are at present typically designed by human designers and optimised for high performance, Thus these circuits typically violate the assumptions made by most formal asynchronous design styles. For example, many formal design styles of do not admit combinational looping behaviours. That is, within the scope of one "evaluation" of a local circuit, no signal changes more than once. An example of a pulse gate circuit that admits looping behaviour is the serialiser in figure

Gate	FO4 delay	Std. Dev	$T_{g,min}$	$T_{g,max}$
Repeater	29.9 ps	2.2 ps	27.7 ps	40.9 ps
Or	31.9 ps	2.7 ps	29.2 ps	45.4 ps
Single condition	31.5 ps	2.7 ps	28.8 ps	45.0 ps
2 condition AND	29.4 ps	2.6 ps	26.8 ps	42.4 ps
2-condition OR	29.1 ps	2.3 ps	26.8 ps	40.6 ps
2 OR w/conditions	33.3 ps	3.5 ps	29.8 ps	50.8 ps
SR latch	37.0 ps	3.3 ps	33.7 ps	53.5 ps

Table 2.1: Gate delays of a typical pulse gate library (from [2])

2.4 (again from [2])., in this case, finite looping behaviour.

2.2.2 Low variance

Pulse gates have low variance, both in terms of the spread of different possible nominal delays of different pulse gates, and also the possible PVT (process voltage and temperature) variation of gates in implementation compared to that nominal. This can be seen in the example timing measurements in table 2.1. Converting to a bi-bounded delay model based on $-1/+5\sigma$, the minimum time is $26.8ps$ and the maximum is $53.5ps$

One reason for this is that the propagation time is largely determined by the reset loop which is common to most of the pulse gates in library, with only the input pull down changing. This contrasts to say static CMOS where the whole gate changes with function.

Thus pulse gate circuits can be designed with timing arcs that are assumed to remain in the order they occur in the nominal case, and further that this nominal-case order is that the would be suggested topologically, i.e. the event that has more gate to it in series occurs later.

This contrasts with, for example, burst-mode design where the output depending on the order of evaluation of the gates would be disallowed, or QDI design where a handshake join would be added to ensure both signals have been computed before being combined.

Un-timed Static Gates

One extension to the construction technique that is allowed is the addition of static gates to combine data levels from a latch before being used in a pulse gate guard. Arbitrary use of static gates particularity to process pulse signals could lead to electrical errors by distorting the usual pulse shape. Even if this were designed for, they would break the low variance assumption the above rules offer. Using static gates only for data signals, combined with the above construction rules ensures the timing can not propagate further than the receiving pulse gate, since that does not derive timing from the data inputs. Effectively the static gate is treated as being part of an elaborate pull-down network for the receiving pulse gate, albeit one that significantly increases the setup time of data signal at that gate.

Production Rules of Pulse Gate Circuits

One gate level description asynchronous circuits, including pulse gate circuits, is a "Production Rule System" [44].These are a set of pairs of input preconditions and the output actions they cause. For level-based asynchronous circuits the output actions are rising and falling transitions. For pulse circuits the production of an atomic pulse is also a possible result of a production rule. Let an "action" be either a pulse or a data edge. Similarly the guard for level-based circuits is a Boolean function of level signals, where for pulse circuits the guard may also include pulse signals.

An event is is an action which is capable of causing further actions. That is, pulses or level transitions capable of causing a production rule or satisfied and thus the gate to perform its action is called an event. The typing rules discussed above for pulse gate circuits mean that signals are either pulse or data signals. By their nature, short atomic pulses do not have state that other gates can include in their guards. The transition actions on data signals are also not allowed to be events by the typing rules. Thus a signal is used either for events or static state (in the production rule sense), never both and never switching between them in a state-dependent way.

For multiple events arriving near-simultaneously at a gate, causality can be determined by applying them one-by-one and seeing which one results in the output action being produced. The result of this procedure should result in the same output regardless of the order. Simultaneous arrival of events that would result in different outputs depending on order is usually a hazard e.g. data change arriving at the same time as a pulse in a pulse AND gate, unless an explicit arbiter is constructed.

2.2.3 Timing Hazards

There are two possible ways in which a pulse-gate circuit can escape its intended function. The first is that an electrical error in the pulse gates can occur, where the assumptions about its input types are violated and cause the pulse output to malfunction, e.g. by producing runt or overly wide pulses, or two output pulses for one input. Logical errors occur when pulse gates function electrically correctly but the system behaviour is different from what was designed. Both these types of failures are produced from the same hazards. Whether the failure is electrical or logical depends on the degree to which the assumption is violated, as show in figure 2.5.

The different types of hazards that can lead to these errors are differentiated by the typing of the signals being combined and the intended order of the combination. There are 6 types of pulse gate hazards, enumerated here and shown in figure 2.6:

- A **Hold** hazard exists when a pulse is ANDed with a data signal, and the pulse arrives and is expected to be combined with the old value of the data signal before the data signal changes. If the pulse arrives later than nominal, or the data changes

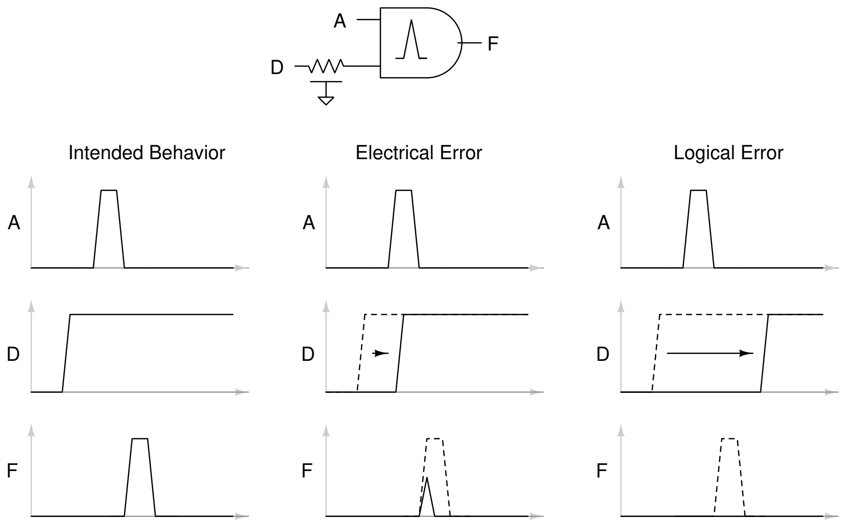


Figure 2.5: Pulse gate errors: logical and electrical

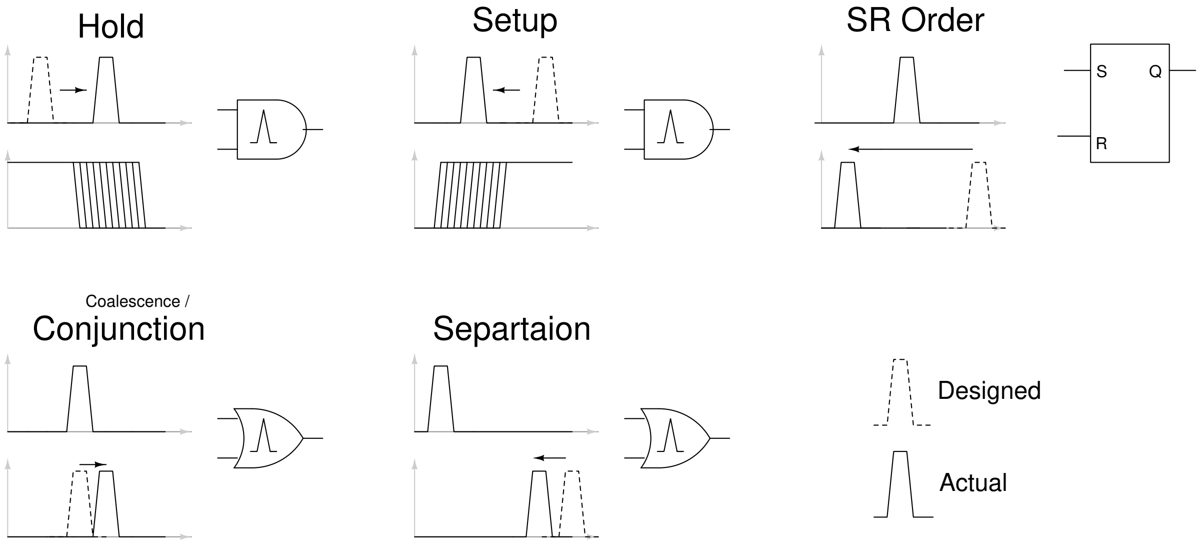


Figure 2.6: Pulse gate Hazards

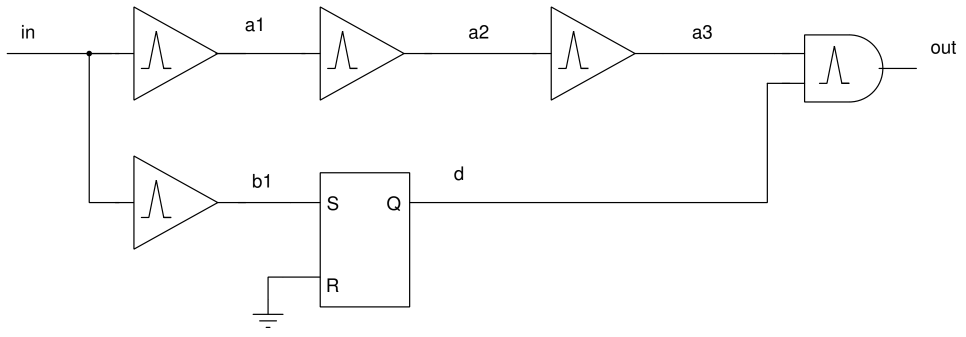


Figure 2.7: Example fragment of a pulse-gate circuit

earlier, a hold violation occurs.

- A **Setup** hazard exists when a pulse is meant to be ANDed with the new value of a data signal, after that data signal changes.
- A **Retrigger** hazard exists when a pulse gate fires for the second time too soon after the first, such that the pulse loop is still resetting.
- A **Set-Reset Order** hazard exists when a SR latch is meant to receive a set and reset activation in a certain order, but that order changes so the latch ends up having the wrong value.
- A **Coalesce** hazard exists when two pulses are meant to arrive at the same time at a gate and coalesce.
- A **Conjunction** hazards is similar to a Coalescence hazard, except for gates where the arrival of both pulses is required to produce the output.

2.3 Timing Constraints

To ensure that the assumed behaviour in the nominal case is what actually occurs, either the circuit must be constructed so that no timing dispersion is possible, which in general is incomplete and where possible is often slower than ideal. Where timing hazards might possibly occur, timing verification must be performed to ensure that for the actual delays that occur in practice, the circuit has the intended behaviour. Considering the circuit in figure 2.7. The interaction between pulse `a3` and data signal `d` at the output gate needs to be constrained to occur in the designer intended order.

$$t(d) + t_{setup} < t(a3)\ at\ out$$

Since pulse gate circuits often have signals which occur many times within a local scope of evaluation, stating such an inequality alone can be ambiguous. If multiple `a3` pulses and `d` changes occur in a local looping fashion, each pair of `d` and `a3` need to follow that order but it perfectly allowed for, after the first `a3`, a second `d` change to occur. If a simple inequality like the above were used it would not be possible to accommodate such behaviours. Solutions such as indexing the first, second, etc., occurrence of the same event are possible but difficult given that which events occur in a pulse gate circuit is data dependant. This is particularly true due to the wide use of one hot encoding.

The solution chosen in this work is to express the timing constraint in terms of path inequalities. If the timing behaviour is truly local, then one recent common ancestor can be identified. The timing constraint can then be expressed as an inequality on the paths from that common ancestor to the signals in question. For our current example `in` is the common ancestor.

$$t(in \rightarrow b1 \rightarrow d) + t_{setup} < t(in \rightarrow a1 \rightarrow a2 \rightarrow a3)\ at\ out$$

In general a pair of signals being compared may have several possible common ancestors and the timing from that ancestor to those two signals needs to be checked. The fact that such common ancestor(s) exists and checking timing from them is sufficient is an assumption that will be elaborated on in the course of this thesis.

The enforcement of timing constraints such of these is generally achieved by characterising each gate type used in the circuit, and then substituting appropriate numbers into the derived timing constraints and checking the inequalities hold. Characterisation tables for this include data on gate propagation times and the required timing margin for the various hazard (for example, the setup and hold times of a Pulse AND gate).

The characterisation need of hazard margins needs to be not only such that the gate operates electrically and logically correctly, but such that it meets the overall assumptions of the model. For example when considering the rising edge setup hazard for a Pulse AND gate, a situation can arise as the rising edge arrives later where it is on the tailing edge of the pulse at the other input, but there is still enough of an overlap that the gate fires. This is the electrically correct behaviour but violates time timing assumptions of the construction rules that timing is based on pulses only. Thus for purposes of characterisation this should still be considered a setup violation.

Chapter 3

The Unit Time Model

Now the "unit time model" is introduced. This is the local part of the model pair proposed in this thesis. First the model itself will be proposed and specified formally. The requirements for it to be valid and applicable are presented. The notion of coherence and how it supports the unit time model is discussed. Lastly how the unit time model can produce timing constraints and how those timing constraints validate the unit time model is introduced briefly.

3.1 Unit Time Model

The unit time model of a pulse gate circuit (or a region of a pulse gate circuit) is an finite state machine model of an asynchronous circuit. The circuit is described as sequences of discrete "states". A *state* is defined as the set of events present nominally simultaneously in the circuit, as well as the value of the data signals at that time. States are separated by the local sequencing of individual gates. The successors of a state are defined as the sets pulses present in the circuit simultaneously one topological, with latches updated if they have accepted pulses. Series chains of pulse gates become a shift

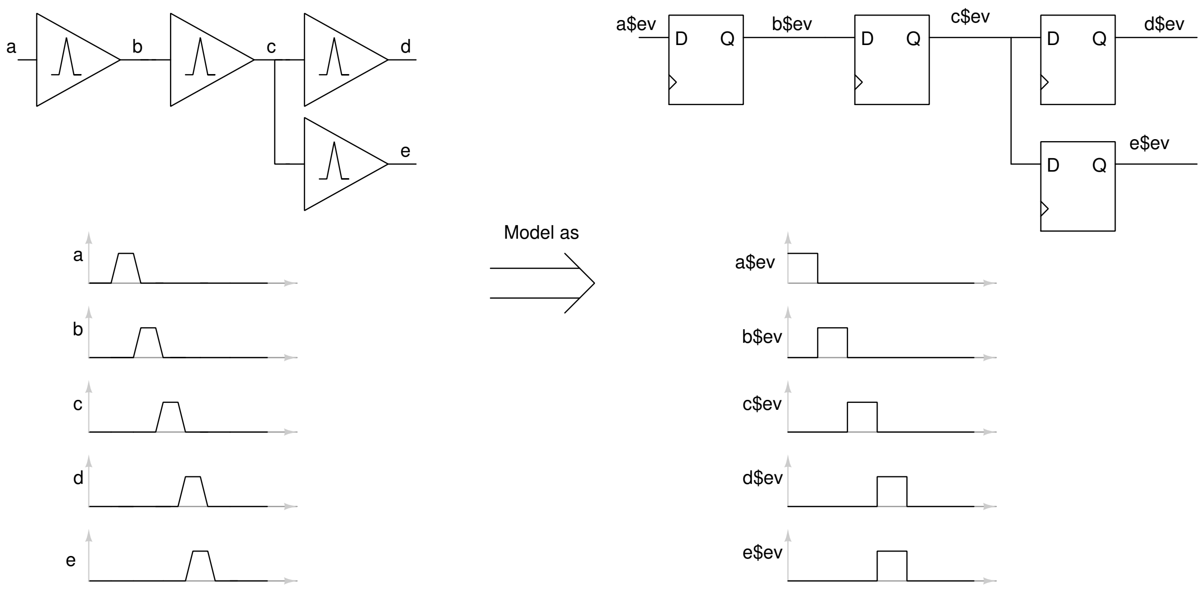


Figure 3.1: Simple pulse circuit and its unit time model

register. An example of this is shown in figure 3.1.

Given a circuit and an input specification, the unit time model through simulation (symbolic or otherwise) can identify a possible behaviour of the circuit, which is what the circuit would do if all gates in the circuit had the same constant nominal gate delay. In this unit delay view, the events occurring in the unit time model are a near-simultaneous group of pulses and the next state is the state of the circuit sampled one unit gate delay later.

By inspecting the behaviour of the circuit under the strong behavioural assumption, actions that need to occur in a certain order to produce that behaviour can be identified. That is the pairs of actions that constitute a hazard as defined in chapter 2 can be identified. A looser assumption can then be made that the same behaviour will be reached provided those key event pairs occur in the correct order, since for gate to behave differently would require a hazard to be violated. The time of the events in the same state in real-time execution are allowed to spread out as long as the eventual behaviour is the same. It is this latter looser behaviour that is actually enforced on the circuit when performing verification.

The unit time model of a gate circuit specified by productions rules is a state machine model based on the topology of the circuit.

Definition 1. *A* ***state*** *in the unit time model is the product of the following factors:*

- *For all signals, whether an action is occurring on that signal for that state. For a pulse system this is the pulses and data changes.*
- *For signals that have a level, the value of that level (after the transitions have been taken into account).*
- *Any internal state of gates, e.g. consensus gates (pulse and regular).*
- *The generating machine of the input.*
- *An output recogniser for output verification, reduced to deterministic form.*

Definition 2. *The* ***Unit Time*** *model is a state machine model of an asynchronous circuit. The states of the unit time mode are as defined in definition 1. For a given state, the successor state is given:*

- *Evaluating all the production rules of the gates to determine which actions occur in the new state*
- *For those level signals where an action occurs, the new value of the level signal applying the above actions*
- *Similarly for internal states, the new values after evaluating the production rules affecting them.*
- *The next state of the input generator.*

- *The appropriate next state of the recogniser.*

Note that most components of the unit time model are deterministic. The input generator in general is non-deterministic to allow exploration of multiple possible executions. The gate actions are deterministic except for arbiters, either designed or accidental.

3.2 Validity and Applicability

The unit time model is ultimately a topologically inspired model, based on the fact that a near-simultaneous group of pulses in the nominal case will produce, one gate later, another set of near-simultaneous pulses. In this sense the unit time model rounds all events in the system into near-simultaneous bunches and simulates this. Even if only the weaker behavioural assumption that the intended order of pairwise timing arcs is the topological order, the unit time model is a good fit since it produces those events in that order.

The validity of assuming the unit time model predicts an actual behaviour the circuit can be made to have depends on the following factors:

- Dispersion that exists does not build up too much. A "coherent" property of circuits will be established to deal with this.
- No gate interaction produces different results between unit time model and real circuit.
- The unit time model is the only activity in the circuit or region where it is being applied.
- The circuit does not break the construction rules already in the nominal case.

The coherence property and timing verification issues will be introduced later in this chapter and expanded later on in this thesis. The non-interference requirement will be discussed in chapter 4.

The other issue is if the unit time model is applicable. That is, if the behaviour the unit time model predicts can actually be enforced on the circuit, and that it is the same behaviour the designer wanted. The unit time model is particularly applicable to pulse gate circuits due to the narrow timing values for a typical pulse gate set discussed in chapter 2.

Note that while the near-simultaneous assumption is superficially similar to the fundamental mode view, the fact that this simultaneity is not enforced, just the actual hazards in the circuit means it is not a particularity good fit. In fact variance much wider than a gate delays worth is allowed to build up provided it does not change the behavioural outcome.

That the circuit follows the construction rules in the nominal case can be determined by the examination of the states reached in the unit time model. If one of the reachable states of the unit time model include a pulse occurring and in the same nominal state a data signal that that pulse is ANDed with changing, then this violation of the rule that data signals are stable when being sampled can be flagged. Similarly data gates with set and reset actions trying to occur simultaneously is a violation since it is in effect an implicit arbiter.

3.3 Coherence Depth

One of the key assumptions of the unit time model is that variance cannot build up indefinitely in the circuit.

Definition 3. *The "Coherence Depth" of a circuit or a region of a circuit is how many* *topological steps signals can travel separately before all signals remaining in a region can attribute their timing back to one common ancestor.*

This coherence depth property is a measure of the amount of variance that can build up can in a circuit. If the coherence depth is small then only a small amount of variance can occur. If it is larger then there is more opportunity for variance to build up, but still bounded. If the circuit is such that the common ancestor cannot be enumerated, then variation can build up indefinitely.

An intuitive understanding can be achieved by considering two pulses from the same unit time state, and injecting a timing dispersion between them when they occur in the actual real-time execution of the circuit. Then considering the resultant pulses a number of unit states equal to the coherence depth later. The coherence depth properly assures that the relative timing between the pulses in that later state will be the same as if no dispersion had been injected.

This occurs because, for the circuit to have that coherence depth proper the pulses in the new state will have picked the same one of the starting pulses as their timing ancestor.The circuit can achieve this coherence in several ways. Paths can terminate thus the timing dispersion on that path is absorbed. Otherwise synchronising elements can be used.

Figure 3.2 shows some examples of how coherence can be achieved. All three have coherence depth 3. `a`, `b` and (where relevant) `c` are pulses in the same unit time state at time k, `d` and `e` are pulses in the immediate successor, `f` and `g` in state at time $k+2$, and `h` and `i` in the state at time $k+3$. All three will absorb an injected dispersion. Circuit (a) does so since only one path back to the inputs remains at timestep $k+3$. In circuits (b) and (c) coherence is achieved by means of consensus gates, though any synchronising gate would suffice. In (b) these form an explicit synchronising layer, in (c) there is no

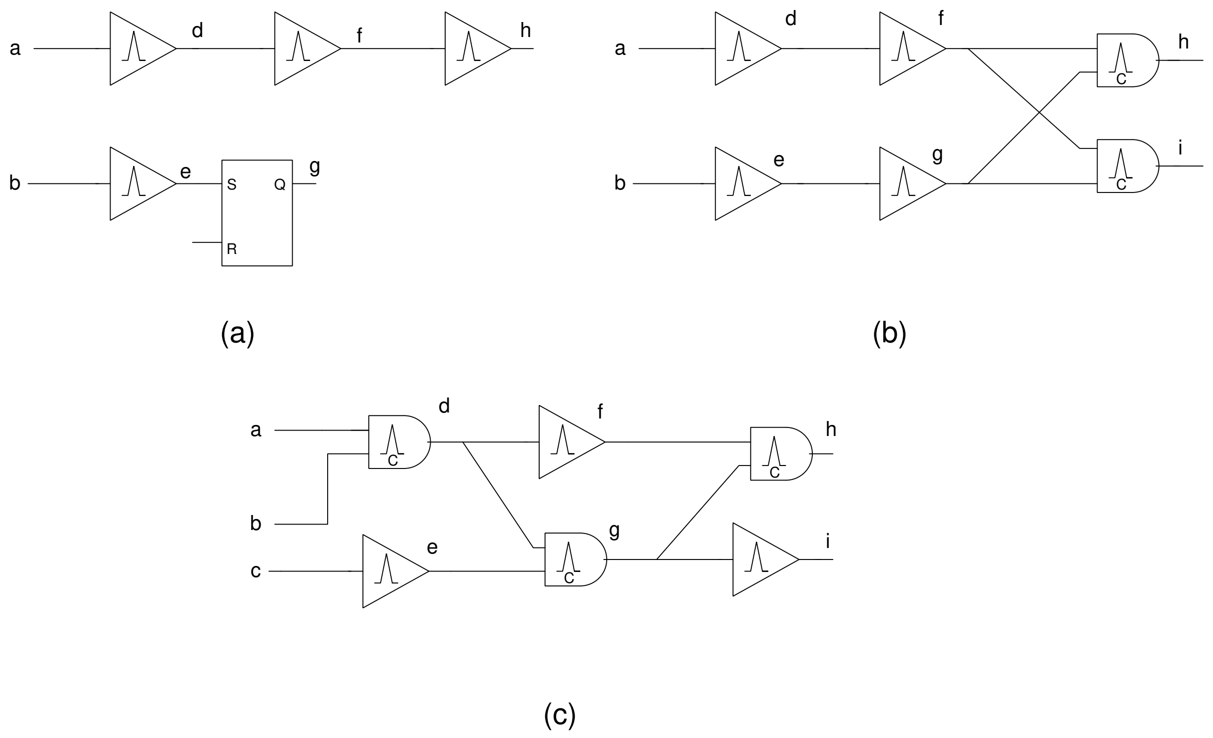


Figure 3.2: Coherence can be achieved by (a) Terminating all but one path (b) Explicit synchronising step or (c) Distributed synchronisation

explicit synchronising layer but enough synchronising elements distributed throughout the circuit that coherence is none the less achieved.

With respect to the unit time model, the coherence depth of a circuit is: starting from any given state how many steps forward before all the resultant actions in the new state have a common ancestor event in the first step, or have timings as if they did. The same definition can be applied to a region of a circuit.

Figure 3.3 shows how the coherence of the circuit from figure 3.2 (c) is achieved considering the real-time occurrences of the actual events. Under each set of nominally simultaneously pulses from the unit time models, the relative time they occur is shown on the graphs. For the first state, a dispersion is introduced arbitrarily into the times of signals `a`, `b` and `c`. Additional dispersion from gate variance is not here considered so the

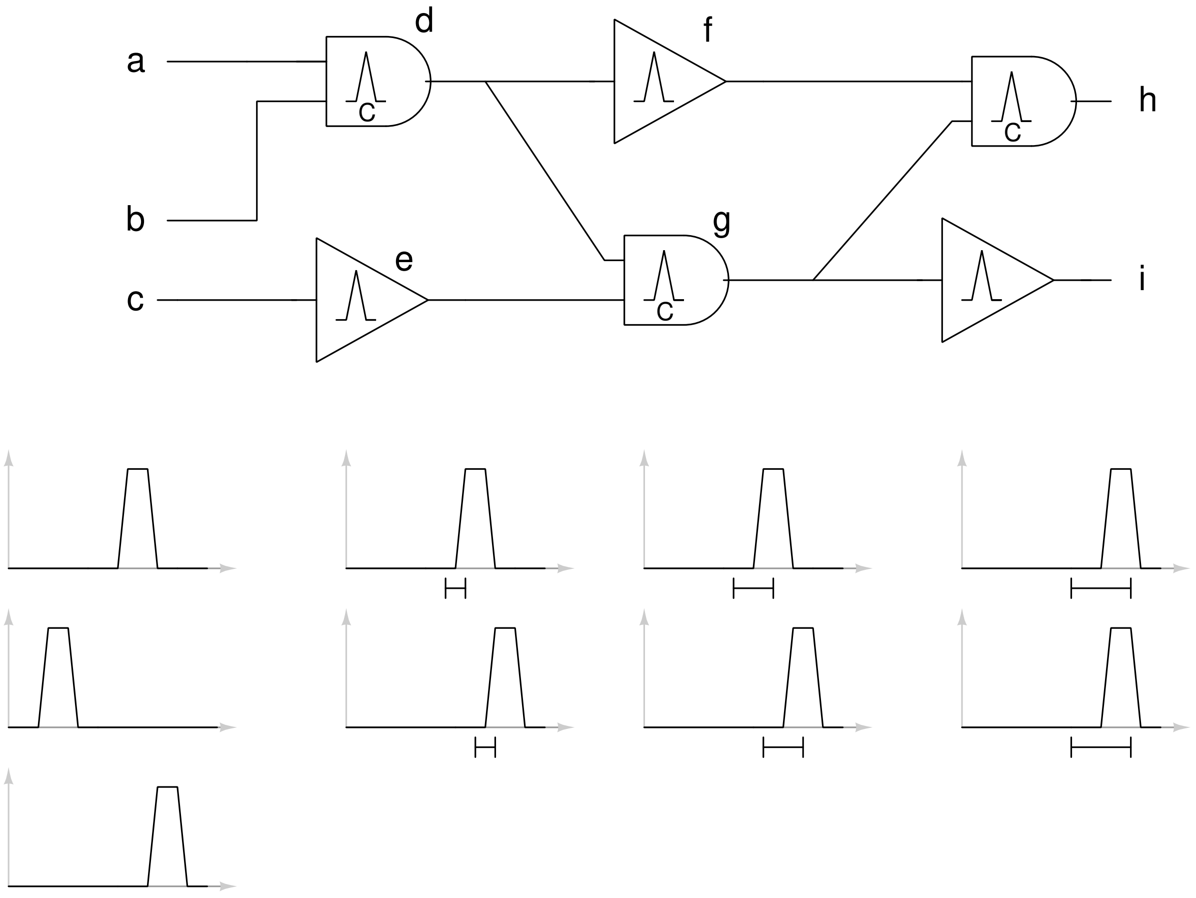


Figure 3.3: Dispersion being absorbed by real time gates

relative times are propopgated. The pulse consensus choose the later of the two input pulse times. Thus by state $k+3$ the two pulses on `h` and `i` occur at the same time.

If the a circuit is such that the behaviour can fork into separate regions which do not communicate with each other for a long time, then the timing dispersion between these two regions will be large, much larger than the timing dispersion within either region. It is possibly unbounded if the regions do not communicate for an indefinite amount of time. In these cases it makes sense to apply the unit time model to these regions individually and resolve the communications between them in another fashion, this will be discussed in chapter 4.

3.3.1 Fast-Slow labelling

The fast-slow labelling is an augmentation of the unit time model to carry information about how variance propagates. This is a discrete model of the variance build-up considered above. A state in this rank-order labelled model consists of a unit time state plus a label added to those signals that have an action occurring in that timestep. Labels where applied are the atoms $Fast$ and $Slow$, where for the pulses in a unit nominal timestep the ones labelled $Fast$ are advancing in notional real time compared to the ones that are $Slow$. Labels are not applied to signals where events are not occurring in a given time step.

The fast-slow labelling is added to trace under the unit time model, starting from some state index in that trace. For the first state of the ranked-order, unique labels are added to each event that occurs in that state. When considering models which receive input, the coherence of the input must be considered. This will be address in chapter 4.

For the next state of the fast- order labelling, the unit time state component is determined as for the regular time model. Gates that produce an action have labelling

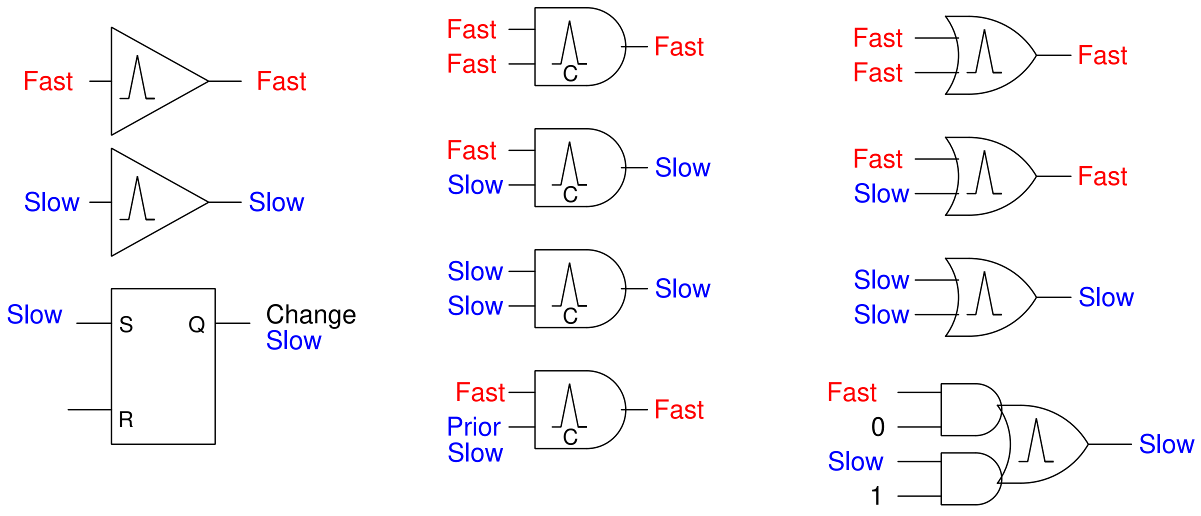


Figure 3.4: Fast-slow labelling rules

as follows:

- For a single incoming event, the produced event has the same label as the incoming event. This could be a buffer where there is only one input, or an OR gate where only one of the inputs fires in the timestep.
- For a pulse consensus gate with both inputs arriving in the same timestep, the output is fast only if all the inputs are fast, otherwise it will be labelled slow. Note that it has no memory of labels, if the gate is activated by one prior and one current event then the label of the current events will be used.
- For a pulse OR gate with multiple arriving timestep, if any of the inputs are fast the output will be fast. Otherwise the output will be slow. This is subject to data guards, labelling is never copied from an input event whose guard is not met.
- Latch changes are labelled similarly.

These rules are illustrated in figure 3.4. The labelling of latches facilitates correct measurement of the amount of dispersion possible on a data signal, and also allows identification of hazards.

3.3.2 Coherence depth measurement

To measure the coherence depth of a circuit, the fast-slow labelling is used to model an injected perturbation. Given a a state in the unit time model, fast slow labels are added as described above. Then the labelled model is simulated forward until all the actions at that later timestep have the same label, *Fast* or *Slow*. If the circuit has the coherent property we need, this labelling will eventually converge to such a state. If it does not, infinite variation build-up is possible and the unit time model does not apply to this circuit.

The coherence depth for this starting step is maximum value this takes over all labellings. This models the perturbation being absorbed as above. This process is illustrated for a simple example circuit in figure 3.5, this example state has coherence depth 3. The coherence depth of the circuit or region being modelled by the unit time model is the maximum of the state coherence depths.

3.4 Single Ancestor Timing Verification

Now that the unit time model has been built, the timing hazards, pairs of events that must occur in a certain order, can be built.

One of the simplest timing verifications that can be performed on a circuit is to check that for each gate the circuit, whenever that gate fires, the fanout of that gate does not create timing hazards in and of itself. This “Single Ancestor Timing Verification” may be achieved in several ways, one way in which this might be enforced is by a set of path constraints as described in chapter 2.

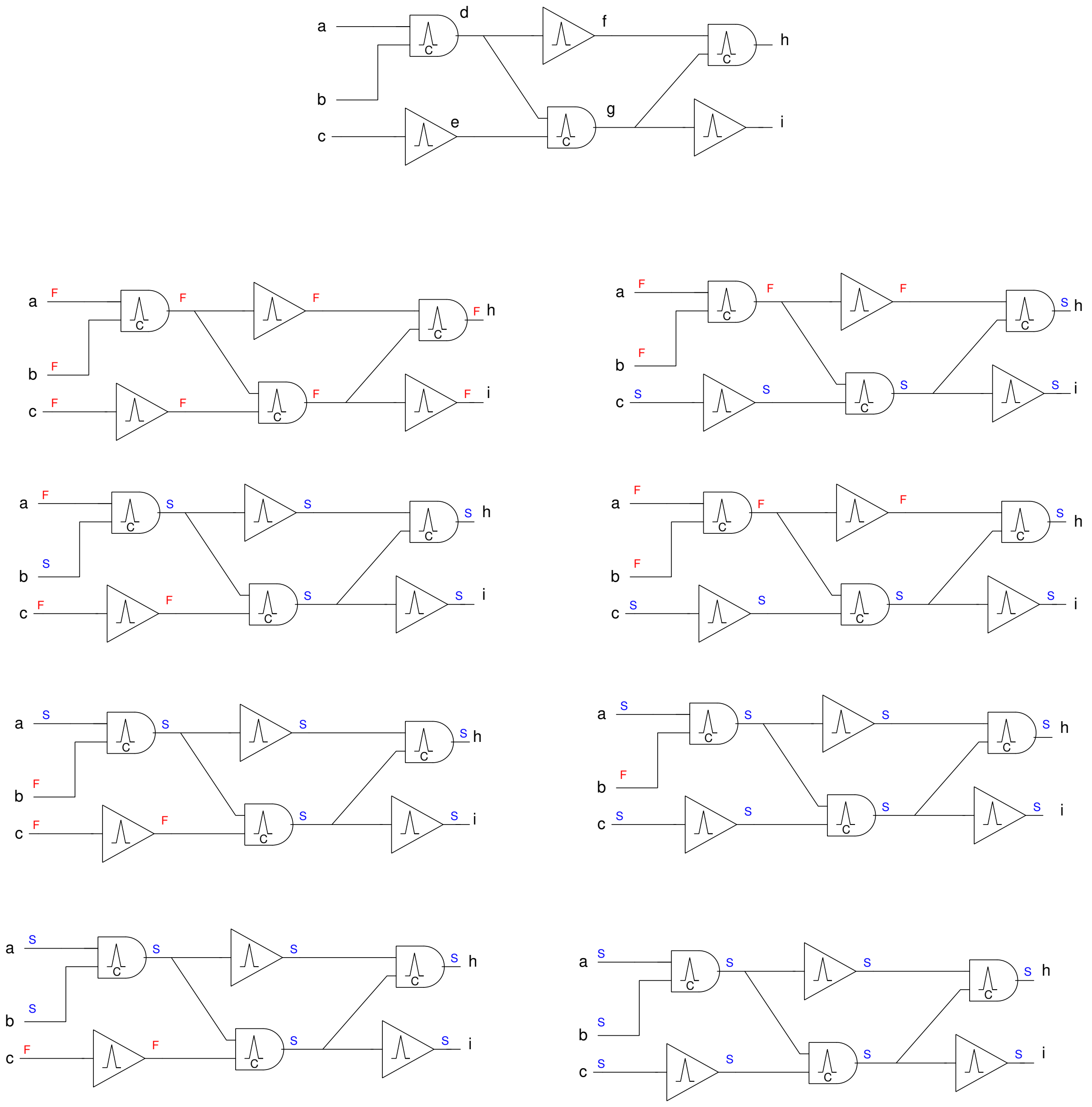


Figure 3.5: Coherence example

From the definition of coherence depth, it can be postulated that only those paths where the shorter “slow” side of the timing path is shorter than or equal to the coherence depth need be considered. This will be proven in chapter 5, as will bounds on longer “fast’ side of the path.

When considering input signals into the circuit, the ‘single ancestor’ may be a pair of input signals possibly even on different timesteps. These have an unknown common ancestor in the previous region. Thus whatever timing model is applied in the current region it may produce a constraint on the input variables that then needs to be checked in the source region of the input.

3.5 Conclusions

In this chapter the unit time model of a pulse gate circuit has been introduced, as has the notion of the coherence depth. The validity of the unit time model has been postulated to rest on the coherence depth, non-interference and timing verification. Chapter 5 will prove that these are sufficient. The concept of coherence depth has been introduced, and how it contains the timing dispersion in a unit time model of a circuit. The requirements of what timing verification needs to be performed has also been discussed.

Chapter 4

Phrases

The applicability of the unit time model is dependant on the coherence depth being small. Trying to build a unit time model of a large system will have an excessively large coherence and it would instead be better to build unit time models of different regions of the circuit and then model the communicating between those regions. Thus it is necessary to extend our unit time model to include communication. Since pulse gate circuits make timing assumptions across interfaces as readily as they do within themselves, it is necessary that coherence also be available across communication. On the other hand, where there are long gaps in communication it does not make sense to preserve coherence and would be better to terminate that coherence reference and start with fresh coherence when the next communication occurs, e.g. in a serialiser system the timing between bits within a word might be within the same coherence but a new coherence for the bits of the next word.

4.1 Phrases

Definition 4. *A "phrase" is a unit timed series of pulses and data-changes that represents one communication interaction between two regions of a circuit. A phrase consists of three parts:*

- *The "cut set", a subset of the signals of the system.*
- *A regular language whose alphabet is logic cubes describing restrictions on the cut set signals.*
- *A phrase coherence depth.*

The "cut set" is a set of signals which describes the boundary over which the communication is occurring. This shows the patterns of unit timed signal events that constitute the communication being described.

Phrase languages are expressed here using a syntax superficially similar to Martin's Communicating Hardware Processes, itself based on Hoare's Communicating Sequential Processes. A signal being mentioned means an action on that signal is expected to occur in that state. A signal not being explicitly mentioned means it is assumed not to occur. The alternation operator "|" means one signal or the other is expected to occur. The parallel combination operator, "||" in traditional CHP, means both signals must occur. In the phrase version "&" is used instead since it similarly means both signals must occur. In the unit timed version used here, the ";" operator is added means exactly one unit time step later, and either a blank state (nothing between two semicolons) or $\varnothing$ describes a unit time step with no events crossing the region boundary.

Note that phrases are still regular languages, or equivalently automata and this is a syntactic choice since that syntax is better suited to the fact that each state in the automata is a requirement on multiple signals. These requirements are cubes, though expressed with a slightly unusual syntax due to their sparseness. Further while pulse events that are mentioned must occur, for data change actions the mentioning of a signal by default means a don't occur, since the producing region changing a data latch may change the data from that value to the same value instead of necessarily toggling it. If needed additional decorations could be added to imply must change or even must rise or must fall.

Each phrase also has a coherence depth. This is the largest distance back between common ancestors of any pair of nominally simultaneous signals in the phrase, or between a current event and the unseen progenitor of a future event. This is at most the coherence depth of the region generating the phrase, or it may be smaller depending on the structure of the region.

For example a simple serial link system (serialiser and deserialiser) can be described with 3 phrases. The first is the "go" phrase to start the serialiser. This has a language of just $d[0:7]; go$. That is the data signals are set up in the first step of the phrase, and the "go" The "done" phrase emitted the deserialiser is similar. In practice the "go" and "done" phrase may also include the setup of the input and output data latches respectively. They would remain one state long but that would have additional signals.

The more interesting phrase is the one describing a word on the pulse-based serial link. Consider a 4-bit word with 4 unit gate delays between each symbol. This described by the language:

$$(ser0|ser1); \varnothing; \varnothing; \varnothing; (ser0|ser1); \varnothing; \varnothing; \varnothing;$$

$$(ser0|ser1); \varnothing; \varnothing; \varnothing; (ser0|ser1);$$

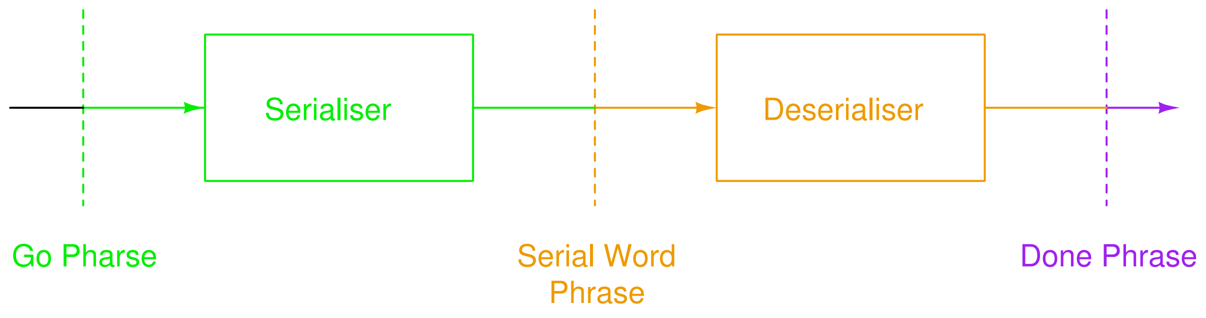


Figure 4.1: Phrases dividing up a SerDes circuit and regions derived from them

4.1.1 Regions by cut

Given a set of phrases for a system given by their languages we can identify the regions associated with each phrase automatically. First the cut set can be obtained by inspection of the language for what signals occur. Using the cut set as a starting point, a topological flood fill up to the boundaries defined by the cut sets of the other phrases. Thus we obtain a region that receives that phrase, computes on it, and emits zero or more output phrases (of the one or more topologically possible output phrases found by flood fill). This flood fill is shown for a SerDes system in figure 4.1

In general this procedure leads to overlapping regions, where two distinct phrases fan out into the same gates. Further this overlap may mean that multiple technically distinct regions may produce the same output phrase or phrases. Thus it is assumed that while a particular phrase is being processed in a region, that the behaviour sponsored by that phrase is the only activity in that region. All phrase activity from other phrases whose fanout region overlaps must occur definitely before or definitely after the phrase under consideration. System-level constructions for enforcing this are beyond the scope of this paper.

4.2 Completeness

To see that a phrase-based model can reasonably model general systems, the following is asserted:

- Control forks can be achieved with a region that has one input phrase and two output phrases.
- Control merges can be achieved by having two different regions produce the same output phrase.
- Control joins can be achieved with two single-pulse phrases whose regions overall and start with a consensus gate. The first pulse to arrive arms that leg of the consensus gate, while not propagating to the rest of the region and thus producing no output phrase. The second then fires the consensus gate and produces further action and an output phrase. Figure 4.2 illustrates these two possibilities.
- Data can be passed in several ways. 1-hot encoding of data is useful for long physical distance communication. Pipeline like data passing can be achieved by having the data as part of the phrase cut. Also phrases can read "global" data from registers assuming that it can be guaranteed that no other phrase that changes that data is co-executed.

Patra and Fussel[41] discuss 5 blocks as being sufficient for delay-insensitive asynchronous systems - Fork, Merge, Mutex, 2x1 Join and Mem. Forks, Merges and Join have been here discussed. Memory can be handled quite flexibly in pulse gate systems with latches in regions but, if desired, a traditional delay-insensitive memory region with 3 input phrases (write 1, write 0 and read) and acknowledgement output phrases could be built. Mutexes are not currently supported by this model as they represent arbitration

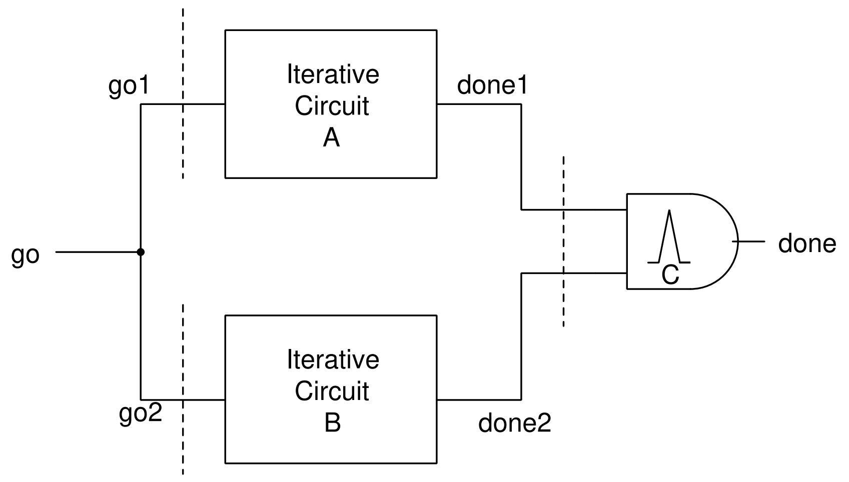


Figure 4.2: System with non-trivial control flow

and the current version of the model does not model arbiters, but given a pulse arbiter they could be constructed quickly.

4.3 Unit time model of phrases

Having determined the regions of the circuit, unit time models can be constructed in each. The input phrase into the region provides the primary inputs that define the behaviour of the region. In the unit time model, a phrase is represented as Moore machine whose outputs are the events and levels of those signals. For the generating machine, the machine is deterministic but is a generator (so supports transitions non-deterministically). The initial state is (arbitrarily one of) the first state of the language. The final state is a trap state which loops forever producing no events.

Knowing the behaviour of a region from this unit time model and its expected output phrases from the flood fill, it can be determined that these are produced correctly. The output recognizing machine or machines are non-deterministic recognisers. They can start in either a waiting state to wait for output, or in the first state of the language for cases where the output begins immediately. An accepting state which loops as long as no unexpected trailing output events are produced is appended.

Phrases also represent a transfer of control, both behavioural and timing across the boundary. Before the predecessor region starts producing its output phrase, control is within that region entirely. When it starts producing the phrase, activity in the successor region can begin. While the phrase is ongoing, the unit time in the successor region is defined with reference to the phrase pulses and thus to the predecessor region. When the predecessor region completes its computation (i.e. goes quiescent), the phrase completes and now control is in the successor region entirely and thus is free to define its own unit time independently.

4.3.1 Coherence depth determination

To allow the coherence depth of a region receiving its input of a phrase to be quantified, labels must be applied to the signals coming out of the phrase. However the phrase specification does not give insight into the nature of the phrase, only phrase coherence depth, which for the input phrase of a region becomes the input coherence depth D_{IN}. Thus in the initial step of the labelling arbitrary labels are applied to all input signals, and this generation of arbitrary labels continues up to an including step D_{IN}. Upon reaching D_{IN}, the coherence depth definition now gives us that whatever the unknown structure of the circuit before the input, those signals now have a common timing ancestor. Thus the signals in this step and later steps all have one common label, the label that was assigned to that unseen common ancestor.

For the output phrase generated by a region, an output coherence depth may be measured considering only the labels of output signals. This may be smaller than the coherence depth of the region as a whole, e.g. if the region latches its outputs.

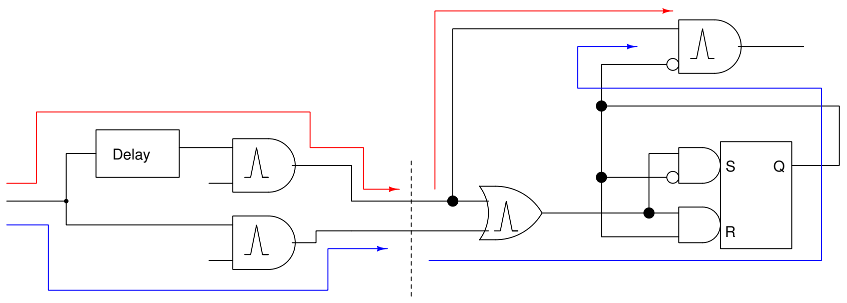


Figure 4.3: Example of a hazard crossing a region

Since the coherence depth of one phrase affects another through the region producing it, a consistent set of coherence depths must be found for all the phrases and regions of the system. This can be achieved by iterative re-computing coherence depths of phrases and regions until an upper fixed point is reached.

The fact that the output phrase might have a smaller coherence depth than the region is important here, as it enables the successor region to have a smaller coherence depth than the current region. Otherwise the coherence depth determination in any looping system would be artificially high or else no upper fixed point may be reached.

4.4 Timing Constraints Across Regions

In tracing back the paths of timing constraints in one region, it may be found that the paths trace back to a pair of distinct pulses in the phrase. If this occurs, the inequality can be rearranged into the form of $t2 - t1 > ...$. Then we can in all possible source regions of that phrase trace all paths and enforce that requirement on $t2 - t1$. Figure 4.3 shows an example of such a constraint in a SerDes setup. The setup constraint between one bit and the next in the deserialiser has its common ancestor in the serialiser.

4.5 Non-interference

Of the four assumptions presented for the unit-time model to correctly model local behaviour in chapter 3, the limited dispersion and different gate behaviour requirements are resolved on a system level scale by propagating coherence depths and timing constraints across phrases as has already been discussed. The requirement for the circuit to meet the construction rules is enforced within the unit time model as before. The remaining requirement that must be addressed is that the unit time model describes all gate activity in that region of the circuit for the time that unit time model is active. Thus consideration must be given to an outer (exterior) model to ensure that combinations of phrases that might co-execute do not use the same gates and thus interfere with each other.

For a given phrase, possible successor phrases can be determined coarsely from topology alone, and execution of the unit time model with output phrase recognisers allows determinations which sets of output phrases can be produced simultaneously. The set of phrases that can co-execute can then be determined using this information.

An execution of a phrase in a region will have a set of signals that fire (for pulses, equivalently change for data signals or internal gate state) at some point in the trace of that execution, and also a set of data signals and internal gate state that is "read" in that execution, that is a pulse occurs where production rules exist that combine that pulse and that state. If it is possible for two phrases to be co-executing then it must be ensured that the no signal is in the fire set of both executions, or the fire set of one and read set of the other. If both regions read the same signal without changing it, no interference occurs.

This determination of read and fire sets must include some behavioural dependence, determining the overall fire and read sets for each phrase and comparing them is too conservative. Behavioural joins intentionally include overlapping regions. If the system is designed so that the overlapping gates are the synchroniser (e.g. consensus gate) and gates after it, no interference occurs in practice. This behaviour is also needed for behavioural completeness as discussed above. However the naive approach of considering all behaviour of the two phrases would flag this behaviour as possible interference.

4.5.1 Speed-Independent Global Model

Given these requirements, a finite state machine global model is proposed. This models a speed independent view of the excitement and resolution of phrases. A state in such a model is for each phrase, whether that phrase is excited or not, and also a designer selected subset of state bits (usually the internal state bits of consensus gates used for global joins). To advance this model, one of the excited phrases is chosen non-deterministically and evaluated. That is, the phrase itself becomes un-excited, output phrases of the region of that phrase become excited. Global state bits that that phrase execution changes in its local model are also updated. This models an atomic execution of one phrase of the system.

By this construction, a phrase that is excited cannot become un-excited except by its own firing. Thus they have the Semi-Modularity property proposed by Muller[39]. This allows the possible circuit behaviour to be viewed as a lattice. When multiple phrases are excited simultaneously then if the system is built correctly, any order of evaluating them will lead to the same resultant state. Thus if a system of phrases has the required non-interference property it is Speed Independent.

For each phrase of the system, global state bits that affect that phrases execution can be determined. The fire and read sets can be determined for each phrase and assignment of the relevant state bits, by considering all possible executions that have that particular state bit precondition. The global state dependent simulation of phrases also allows a refinement of the possible successor phrases. Using this a reachability analysis can be performed to determine which phrases can be co-executing.

From this speed-independent model, there are two possible sources of violations (possible interference between regions) that can be determined:

- If two phrases can be excited at the same time, the fire set of the two overlap or the fire set of one overlaps with the read set of the other.
- If a phrase is already excited once and a second phrase is excited that would, if executed, attempt to re-excite the first phrase. This indicates two overlapping executions of the same phrase is possible

One consideration of real time that must be checked in addition to this otherwise speed independent model is that checks for a given unit time region continues for significant time after starting its output phrase, its direct descendants that could occur while it continues executing must be checked for overlap. Some notion of time must be included in this, else only acyclic system level structures would be permitted by this model. The speed independent check covers all parallel cases because phrases are allowed to evaluate as soon as they become excited or wait arbitrarily long. Thus if it is possible for them to overlap they will in some one of the non-deterministic evaluations of global state both be excited at the same time.

Thus the speed independent model plus this direct descendant timed checking is sufficient to verify the system. Any interference between two parallel phrases will be flagged by the speed independent model, and the timed checks describe check between a phrase and its descendent (where the non-atomicity of phrases isn't covered by the speed independence). It is conservative because it will flag possible errors which have been excluded due to timing. If a more precise model is required, a version of this model based on timed automata could be constructed to include more real time information.

4.6 Conclusions

In this chapter the notion of a “phrase” has been defined as a model of unit-timed communication. It has been discussed how the local unit timed regions can be defined from these phrases, with both the topology and behavioural being algorithmically determined. The notion of coherence depth has been extended to include the coherence depth of phrases, and the interaction between region and phrase coherence depths has been discussed.

By considering which phrases produce each other, the requirements for an outer system level model are detailed. This model allows the non-interfering requirement of the unit time model to be validated. A speed-independent outer model is outlined, with the non-interference property leading to semi-modularity.

Chapter 5

Timing Containment

In chapter 3 is was conjectured that the actual behaviour of the circuit is the same as the behaviour of the circuit predicted by the unit time model, provided the assumptions of that model are met. Those assumptions are measurable coherence depth, non-interference and lack of direct violations, and also that a single ancestor timing verification has been performed. In this chapter this result will be proven. First a the formal model for the "actual" behaviour of the circuit will be specified as a bi-bounded model. This will allow a precise meaning to be given to behavioural containment. The main result will be established by a two-part induction where behavioural correctness leads to timing correctness and timing correctness leads to future behaviour correctness. The proof itself is constructed to be applied to all production rule systems, the examples that are given for pulse gate systems.

5.1 Bi-Bounded Delay Model

A bi-bounded delay model is used. In this model each gate g, upon its production rule becoming satisfied by an event at time t_{in} it produces its output action (pulse event or data change) at time t_{out} such that $t_{in} + t_{min,g} < t_{out} < t_{in} + t_{max,g}$. When considering the bi-bounded model in conjunction with the unit time mode, for an action on signal g at timestep k in the unit model, let $t(k, g)$ be the time that the corresponding event occurs in the real time model.

For gates with multiple input events arriving near-simultaneous acting to produce one output action, there are two possible behaviours. A *delay maximizing gate* is a gate where the gate is activated by the last arriving of its input events. Thus for two input events t_1 and t_2 then $t_{in,eff} = max(t_1, t_2)$. Let such behaviour in general be referred to as a delay-maximising gate, and $t_{in,eff} + t_{min,g} < t_{out} < t_{in,eff} + t_{max,g}$ as above. Examples of this behaviour in the pulse gate family is a conjunction gate, or a consensus gate when both its inputs are arriving near-simultaneously.

The other possible behaviour is a *delay-minimizing gate* where the the gate is activated by the first event. Thus $t_{in,eff} = min(t_1, t_2)$. Examples of this behaviour in pulse gates is a pulse OR gate performing coalescence.

$$T_{min} = \min_g t_{min,g} \quad \text{and} \quad T_{max} = \max_g t_{max,g}$$

The “Maximum hazard time” of a gate is the amount of time after an action arrives at that gate that another action can definitely safely be accepted. For a data change action into the gate, this is the setup time before an event can be accepted safely. For pulse events, there are several hazards to consider - hold time before a data change can occur and also coalescence etc. Usually the longest will be the retrigger time, that is from one event which causes the gate to fire to another event that could. The maximum hazard time of the system is the largest of the maximum hazard times of all the gates in the system, or equivalently the maximum of the required margins of all types of hazard of all gates in the system.

$$t_{hazard\_max,g1} = \max(t_{su,g1}, t_{hold,g1}, \dots)$$

$$T_{hazard\_max} = \max_{g} t_{hazard\_max,g}$$

As discussed earlier the circuit having coherence depth D and making the single ancestor approximation gives that for two events in the same timestep:

$$|t_2 - t_1| <= D * (T_{max} - T_{min})$$

Where D is the coherence depth. Consider a hazard between $g1$ and a later event on gate $g3$ which is resulting from $g2k$ time steps later.

$$t2 - t1 <= t1 - D * (T_{max} - T_{min}$$

$$t3 - t1 <= T_{min} * k + D * (T_{max} - T_{min})$$

To guarantee the hazard does not occur:

$$t3 - t1 >= t_{hazard}$$

If a sufficiently large number of steps is chosen this is always true for any hazard in the circuit. Let such a number of steps be called the order depth D_Ord

$$D * (T_{max} - T_{min}) + D_O * T_{min} >= T_{hazard,max}$$

$$D_{Ord} = ceil\left(D * (\frac{T_{max}}{T_{min}} - 1) + \frac{T_{hazard,max}}{T_{min}}\right)$$

Thus the timing constraints that need to be verified to limit the unit time length are bounded in length. Only paths where the slow path is of length up to the coherence depth D and fast path is of length up to the coherence depth plus the order depth $D + D_{Ord}$ need to be considered.

5.1.1 Bi-Bounded Model of Phrases

For the events in the input phrase, the phrase is generated by a timed extension of the usual generating state machine. It takes each state transition internally for some time in between T_{min} and T_{max}. Let this state time represent the earliest time of the events in the generating machine, noting this event may not be visible at the region boundary. For a input phrase being generated with coherence depth D_{IN} then the visible events of the phrase are generated in the bi-bounded model with times:

$$t_{state} < t_{event} < t_{state} + D_{in} * (T_{max} - T_{min})$$

This follows from the general coherence depth timing bound. The input timing model may provide additional timing guarantees between specific pairs of signals, either measured by another instance of this theory or specified by another means.

For phrases arising from other parts of the system, that this model is followed can be established by applying this theory to that predecessor region. For primary inputs it must be established separately.

5.1.2 Wire Delay

The proof presented here does not include wire delay explicitly. However, it can be made to apply to a model with wire delay by updating the gate times to include the wire delay (assuming a single source for each wire). Let the occurrence time of a signal in the bi-bounded model refer to the time the gate is present at an output. The minimum of propagation times of each gate are updated by adding the minimum and maximum time of the incoming wires into that gate. For the hazard of the gate, a conservative estimate can be estimated by adding the wire times for the fast path but not subtracting it for the slow path, assuming wire delay will never be negative.

$$t'_{min,g1} = min_{g2 \in fanin(g1)} \left(t_{w,min,g2 \to g1} + t_{min,g1} \right)$$

$$t'_{max,g1} = max_{g2 \in fanin(g1)} \left(t_{w,max,g2 \to g1} + t_{max,g1} \right)$$

$$t'_{hazard\_max,g1} = \max_{g2 \in fanin(g1)} \left(t_{w,min,g2 \to g1} + \max(t_{su,g2}, t_{hold,g2}, \ldots) \right)$$

The system wide quantities including the order depth can now be recomputed using these new values.

5.2 Ranked Order Labelling

The ranked-order labelling is a augmentation added to the unit-time state machine, like the fast-slow labelling discussed above. While the fast-slow labelling is useful for intuitive reasoning about coherence, to prove the results that are to be proved a more information dense representation will be useful. A state in this rank-order labelled model consists of a unit time state plus a rank label $l_r(g)$ added to those signals that have an action occurring in that timestep. Rank labels are small positive integers, with lower values representing variance earlier in time and larger values later. Labels are not applied to signals where action are not occurring in a given time step.

The ranked-order labelling is added to trace under the unit time model, starting from some state index k_0 in that trace. For the first state of the ranked-order, unique labels are added to each event that occurs in that state. When considering models with an input phrase, there are three kinds of signals which must be labelled uniquely.

- The internal actions of the circuit and the input actions that occurs in the start-of-labelling state
- Input actions in the next next $D_{IN}-1$ states. Here D_{IN} is the coherence depth of the input phrase.
- One label is assigned to all input actions thereafter.

If the unit time trace is being considered together with a real-time trace from the bi-bounded model then $l_r(k_0, g_1) < l_r(k_0, g_2)$ iff $t(k_0, g_1) < t(k_0, g_2)$. If a unit time trace is being considered in isolation, all possible orderings are considered.

For the next state of the rank order labelling, the unit time state component is determined as for the regular unit time model. If gate g_n produces an action:

- For a single causal event g_1 activating the gate, $l_r(k, g_n) = l_r(k-1, g_1)$
- For a delay-minimising gate g activated by two potentially causal input events g_1 and g_2, $l_r(k, g_n) = min(l_r(k-1, g_1), l_r(k-1, g_2))$
- For a delay-maximising gate with input g_1 and g_2, $l_r(k, g_n) = max(l_r(k-1, g_1), l_r(k-1, g_2))$

5.2.1 Equivalence Between Labellings

While individual labelled traces carry different information, with the ranked ordering carrying more, ensemble of each kind of trace contain the same information. Where one specific ranked order labelling L_r is being considered, multiple passes of the fast-slow

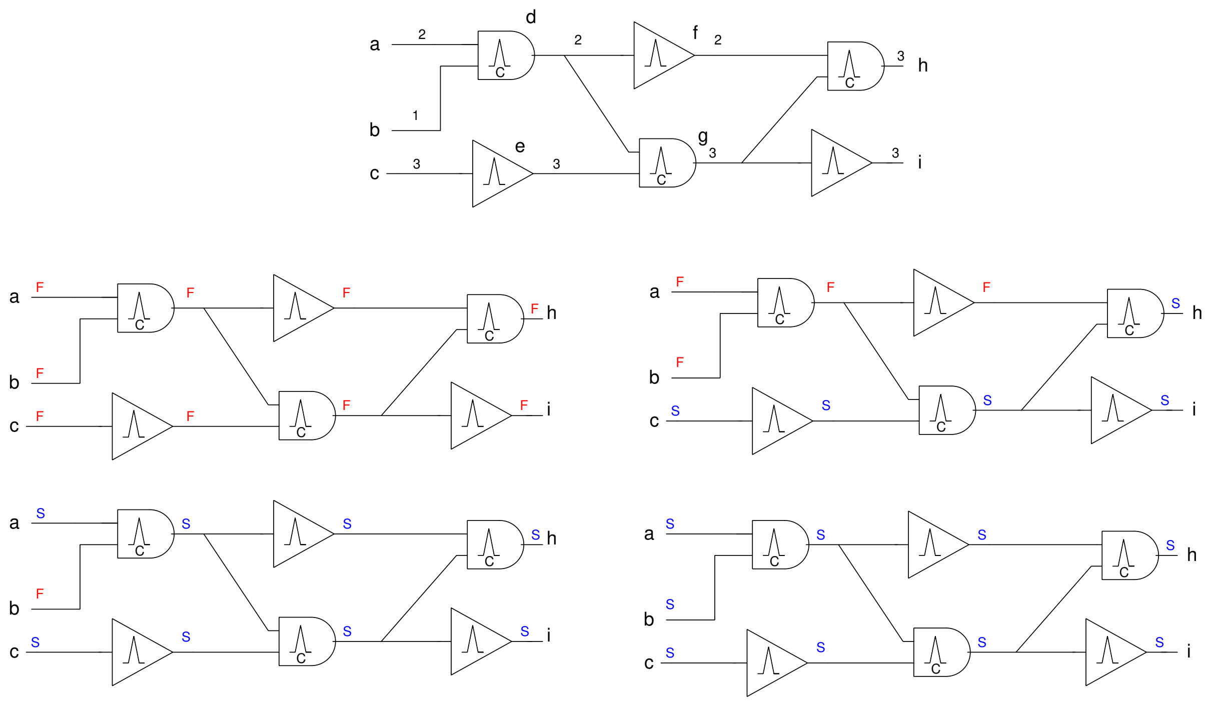


Figure 5.1: Equivalence between ranked order and fast-slow labellings

labelling can be applied to this . Let $l_{f,i}$ be a fast-slow labelling where $l_{f,i}(k_0, g) = Fast$ if and only if $l_r(k_0, g) <= i$. Then for $k > k_0$ we can determine the ranked order label. If $l_{f,i}(k, g) = Slow$ and $l_{f,i+1}(k, g) = Fast$ then $l_r(k, g) = i$. This process is illustrated in figure 5.1. Where all possible ranked order labellings are being considered, the same information can be obtained from considering all possible fast-slow labellings applying this procedure repeatedly.

Note the equivalence between the two labellings means results derived with one apply to the other. Coherence depth can be measured with ranked order labels similar to fast-slow labels and the circuit will have the same coherence depth measured both ways. This equivalence allows the unit time model to be proven here with the rank-order labelling abut computer implementation to rely on fast-slow labelling and have the proven results apply.

5.3 Proof Introduction

Consider a real-time trace T_B from the bi-bounded model. It is required to show that the behaviour of any such real-time trace is contained by the unit time model. Consider also a unit-timed execution trace T_U from the unit timed model started with the same initial conditions and making the same decisions in the non-deterministic input generator.

Definition 5. *A discrete time trace T_U from the unit time model and a real-time trace T_B from the bi-bounded model are said to* ***correspond*** *if:*

1. *A one-to-one mapping can be made between the actions that happen in the bi-bounded trace and the events that happen in unit time trace.*
2. *For two actions $t1$ and $t2$ from the real time model that are mapped to actions in the same time-step of the unit time model, $|t2 - t1| < D * (T_{max} - T_{min})$.*

A pair of traces can be said to be in correspondence for all time, or as a step towards establishing that full result they can be said to be in correspondence up to a given time-step n. Similarly a trace can contain another for all time, or up to time step n as a partial result.

If they are in correspondence for all time, the desired behavioural containment is achieved.

Theorem 1. *Given a circuit C with finite coherence depth D that satisfies single-ancestor timing constraints, any pair of unit time trace T_U and bi-bounded trace T_B from the same initial conditions and where the input generator makes the same non-deterministic choices are in correspondence for all time.*

This will be established by induction. First, the base case will be built. Then the induction will be established in three theorems.

1. If the unit time trace contains the bi-bounded up to step $n-1$ and the one-to-one action mapping exists for step n, then step n also satisfies the timing bound and thus the bi-bounded trace corresponds for step n.

2. Further to that general timing bound, it will be shown that the possible values between two signals is bounded by the actual delays of gates fanning in to those signals as opposed to the global minimum and maximum

3. Finally it will be shown that given these results a one-to-one mapping can be built for the actions in unit time step $n+1$.

Theorem 2. *For a circuit C, any pair of T_B and T_U are in correspondence for $n = 0$ and a one-to-one action mapping exist for timestep $n = 1$.*

Proof. Consider the actions input into the circuit region and those internal to the region separately. At $n = 0$ the generating state machine for both the unit-time and bi-bounded produce the same actions, since the bi-bounded generating machine is defined by addition of times to the unit-time one. Thus input actions correspond. The unit-time model has no actions at timestep 0. The bi-bounded model similarly has no spontaneous internal actions at the start, the first internal actions are those sponsored by the inputs, which will later be corresponded with timestep 1. Thus the same actions exist in that timestep. □

5.4 General Timing Bound

First knowing that the actions in the current unit time step exist behaviourally in the bi-bounded model, it will be shown that those actions obey the general coherence timing bound. This occurs since the circuit is known by precondition to have a finite measured coherence depth, and the same synchronisation elements that captured the labels in the labelled unit time model will capture actual delay. This is trivially true considering only the initial dispersion and not the dispersion collected along the paths. When considering on-path delay it is possible that this might cause two actions to swap order in real time compared to label. However if we consider the ranges of real time possible for each label in each step, it can be seen that these ranges follow the labels since the inversion of actual times requires the two events that swapped to have time within each others ranges.

Consider the example in figure 5.2. Part (a) shows that capture in nominal time occurs as expected, and the ranges of spread of those signals due to on-path dispersion is shown. Part (b) shows that adding on-path dispersion in some cases causes no issue as before. In part (c) at the inputs to gate `h`, the timing of its inputs have shifted enough that the earlier labelled input `f` is later in time and thus the timing on `h` is based on that and not on `g` as predicted by the labelling. However this still results in output timing which is within the range predicted from considering `g` as the input and thus is still acceptable.

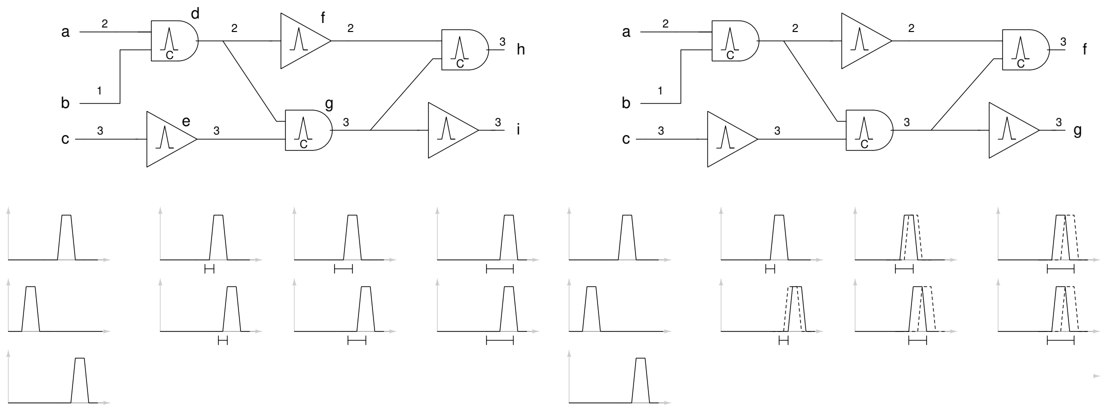
Theorem 3. *Suppose that T_B and T_U have coherence depth D, corresponding actions for timesteps $k = 0..n$ and the delay bound from theorem 1 applies for timesteps $k = 0..n-1$.*

Then for two actions $t1$ and $t2$ in timestep n, $|t2 - t1| < D * (T_{max} - T_{min})$.

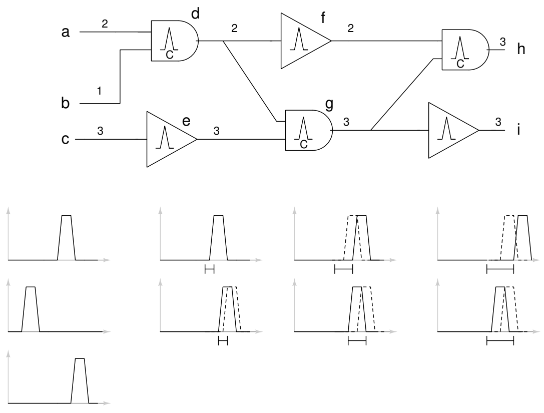
Proof. Let $k_0 = n - D$. Let T_R be a rank-order trace derived from T_U with the rank orders applied at timestep k_0 in conformity with T_B. It will be shown by induction for $k >= k_0$ that if $l_r(k, g) = r$ and $t_r = t(k0, g_2)$ such that $l_r(k, g_2) = r$ then

$$t_r + T_{min} * (k - k_0) <= t(k, g) <= t_r + T_{max} * (k - k_0)$$

For $k = k_0$, all signals have distinct labels and this statement collapses to $t(k, g) <= t(k, g) <= t(k, g)$ which is trivially true.



(a) Nominal case (b) Dispersion added



(c) Dispersion resulting in flip

Figure 5.2: Pulses may disperse in various ways but the ranges of possible values follow the labelling

Assuming the bound is true for all gates up to timestep $k-1$, consider a gate g_n in timestep k, there are have 3 cases:

1. If g_n has a single exciting event g_1 causing its behavior, then $l_r(k, g_n) = l_r(k-1, g_1)$ and $t(k-1, g_1) + T_{min} <= t(k, g_n) <= t(k-1, g_1) + T_{max}$. Thus

$$t_r + T_{min} * (k - k_0 - 1) + T_{min} <= \\ t(k, g_n) <= t_r + T_{max} * (k - k_0 - 1) + T_{max}$$

2. If g_n is a delay-minimising element for two input causal events g_1 and g_2. Let $t(k-1, g_1) < t(k-1, g_2)$. Thus $t(k-1, g_1) + T_{min} <= t(k, g_n) <= t(k-1, g_1) + T_{max}$ and thus:

$$t_{r1} + T_{min} * (k - k_0) <= t(k, g_n) <= t_{r1} + T_{max} * (k - k_0)$$

   If $l_r(k-1, g_1) <= l_r(k-1, g_2)$ then $l_r(k, g_n) = l_r(k-1, g_1)$ and this is directly what is required. If instead $l_r(k-1, g_2) < l_r(k-1, g_1)$ then $l_r(k, g_n) = l_r(k-1, g_2)$. The case assumption implies $t_{r2} <= t_{r1}$. Thus $t_{r2} + T_{min} * (k - k_0) <= t_{r1} + T_{min} * (k - k_0) <= t(k, g_n)$

   For the other side of the inequality, consider $(k-1, g_1) < t(k-1, g_2)) <= t_{r2} + T_{max} * (k - k_0 - 1)$ and thus $(k, g_n) < t_(r_2) + T_{max} * (k - k_0 - 1) + T_{max}$ giving the desired inequality in the second case.

3. If g_n is operating as a delay-maximiser of two input events g_1 and g_2 then a similar argument applies in the opposite direction.

By the definition of coherence depth, we know that all $l_r(k, g)$ are equal for $k >=$

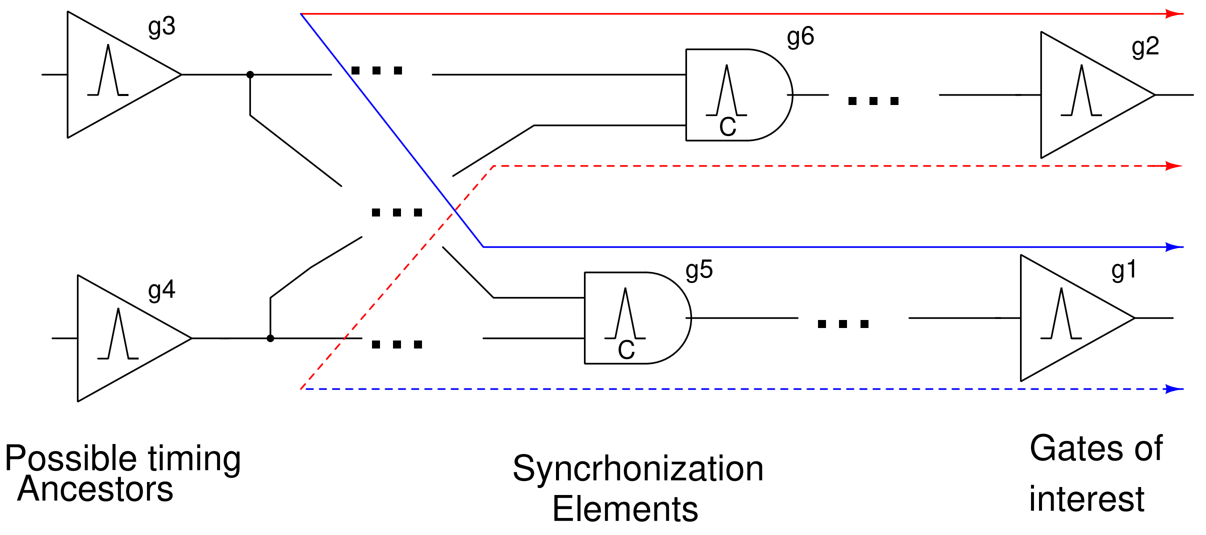


Figure 5.3: Timing based on single ancestor

$k_0 + D = n$ then for $k = n$ we have that

$$\exists t_r \forall_g \Big(t_r + T_{min} * D <= t(k,g) <= t_r + T_{max} * D \Big)$$

Thus any two actions in the state n have real times that are at most $D * (T_{max} - T_{min})$ apart. $\square$

5.5 Specific Timing Bound

Now consider the timing separation between two pulses. From theorem 3 the weaker result that $t_2 - t_1 <= D * (T_{max} - T_{min})$ has already been established. Now it will be shown in fact this is bounded to be based on the specific paths from single ancestors into that gate. An example of this is shown in figure 5.3.

Theorem 4. *Assuming T_U and T_B are in correspondence up to time step n, then for two events $g1$ and $g2$ in timestep n at times $t1$ and $t2$, over all possible ancestors $g3$ in* *timestep k_0 then:*

$$t2 - t1 <= min_{g3}\Big(t_{max}(g3 \rightarrow ... \rightarrow g2) - t_{min}(g3 \rightarrow ... \rightarrow g1)\Big)$$

Proof. From the definition of coherence depth, $g1$ and $g2$ have a common labelling ancestor $g3$ in timestep $k0$. Let $g4$ be another event in the starting timestep. Supposing the label of $g4$ is lower numerically (earlier in time) than that of $g3$. The fact that the label of $g4$ is eliminated means one of:

- There exists two delay maximising elements on the paths from $g4$ to $g1$ and $g2$ that chose the label of $g3$ over that of $g4$,
- $g4$ does not have a path to influence the labelling of either $g1$ or $g2$.
- A mixture of delay minimising and maximising elements exist that always choose the label of $g3$ over that of $g4$

For the delay maximising case, let $g5$ and $g6$ be the delay maximising elements on the paths to $g1$ and $g2$ respectively, from both $g3$ and $g4$. If for the times in the bi-bounded trace $g5$ and $g6$ both choose $g3$ then we have our result with $g3$ as the chosen ancestor gate. Note that any decrease in time $t4$ of the event on gate $g4$ has no event it moves the consensus gates further into the region where they choose $g3$. If the time $t4$ increasing causes $g6$ to choose $g4$ as its ancestor instead, the inequality remains true since $t2$ is increased making the inequality looser.

If $t4$ increasing causes both $g5$ and $g6$ to choose $g4$ instead of $g3$ as their answer the inequality still holds but now with $g4$ as the ancestor gate. If $t4$ increasing causes $g5$ to choose $g4$ but $g6$ to choose $g3$ then again we have our inequality still holding with $g4$ at the common ancestor and $t2$ for that case reduced by the fact that $t3$ pulled $g6$ earlier.

In the case where the label of $g4$ does not affect the labels of $g1$ and $g2$ due to the lack of a path, the same lack of path prevents the actual timing from affecting $t1$ and $t2$, so we have our result directly. For the mix of minimising and maximising cases, if these elements are arranged to always choose the label of $g3$ then they will also always choose its timing.

For cases when the chosen label of $g3$ is earlier than $g4$ and still chosen, this indicates the presence of delay minimising elements and the same argument presented for delay maximising elements applies for delay minimizing elements instead, except that it is $t1$ that can get earlier in time and still satisfy the version of the inequality considering $g3$, if $t2$ gets early we have to switch to considering the $g4$ version of the inequality.

□

5.6 Behavioural Correspondence

Now that the timing bounds between events in the current state has been established, it will be shown this and the local timing verification that is a precondition of the proof combine to establish that the behaviour of the next unit time step is replicated in the actual circuit and thus the one to one mapping of actions can be established.

Theorem 5. *Assuming traces T_B and T_U are in correspondence up to timestep n, then a one-to-one mapping exists for actions in timestep $n+1$*

Proof. For each gate g_n in timestep $n+1$, the unit time model makes a prediction that that gate is about to fire in the next step, and any current step events that could cause the next gate to fire (under some non-causal preconditions).

For each of those events, for each data precondition g_2 in those guards, to ensure the bi-bounded model to have the same behaviour it is require $t_{g_{2},n} >= t_{g_{1},n} + t_{su,g_n}$ and $g_2, n <= t_{g_{1},next} + t_{hold,g_n}$. This is illustrated in figure 5.4.

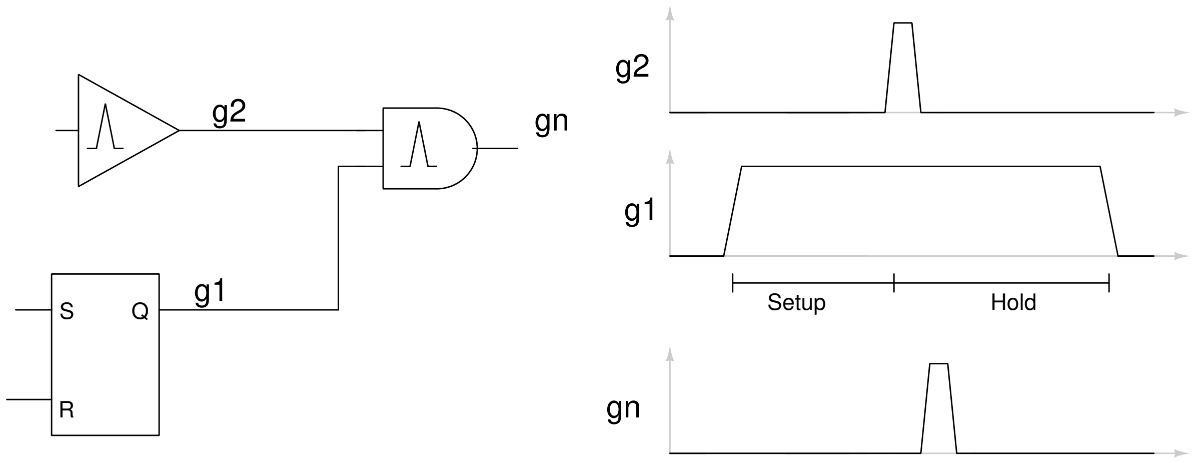


Figure 5.4: Timing based on single ancestor

For the setup constraint, the constraint can be met if $t_2 - t_1$ is constrained to the maximum of its single ancestor value by theorem 4. Thus the single ancestor timing verification database directly checks this.

For the hold constraint, there are two possibilities. One is that the next event to change g_2 is the same next event as predicted by the unit time model through a path predicted by the unit time model. Thus, like the setup case, this is covered by timing constraints. The other is the hypothetical creation of an unexpected earlier event on g_2 but this requires a timing failure at the inputs of g_2 which would be covered by timing constraints.

Thus for each g_1 and g_2 driving gate g_n the single ancestor timing model is sufficient to ensure that the gate performs the same in the bi-bounded model as predicted by the unit time model. $\square$

5.7 Proof Conclusion

Now the proof of **Theorem 1** can be concluded by application of the sub-theorems to achieve the induction proposed earlier.

Proof. By induction. Theorem 2 gives us correspondence for $n = 0$. Applying theorems 5 and 3 in turn gives us that if correspondence exists for timestep n then it exists for timestep $n + 1$. $\square$

5.7.1 Output Phrase Conformance

For each output phrase the region produces, the output phrase may have a coherence depth D_{out} that is smaller than that of the whole region, for example if the region includes extra latches to buffer its outputs. This can be obtained by a similar labelling process to see how long the output signals settle down to the same label the whole system will eventually settle down to. That the actions in the output phrase exist in the real-time model exist follows from the fact that those actions occur in the unit time model, as checked by its output recogniser, and the same events occur in the unit time model and real time model by the overall result.

That the times between output pulses are proportionally smaller to their reduced coherence depth can be seen by a similar argument to theorem3.

5.8 Conclusion

Earlier in this thesis it was conjectured that single ancestor timing verification was sufficient to contain the behaviour of the circuit to that seen in the unit time model. That result has now been proven. First what is meant by the real time behaviour of a pulse gate circuit is formalised. Then a multiple part induction has been performed alternating between showing how behavioural correctness leads to two timing bounds and how those timing bounds lead to behavioural correctness. Issues of wire delay, and the coherence of output phrases of a region have also been discussed

Chapter 6

Tool Implementation

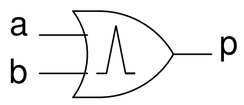
It has been established thus far the the unit time and phrase models reduce the requirements to ensure correct behaviour of the system to several aspects. In this chapter we present a computer tool which verifies the coherence depths of the regions of the system, and performs local timing verification. This is achieved by production of a set of path based timing constraints which can be checked post-layout.

The analysis steps performed are

- From the given phrases and the system schematic, deriving the unit-timed regions
- Determining which output phrases they produce.
- Measuring coherence depths for regions in the system.
- Identifying possible hazards in the system.
- Tracing timing paths that lead to those hazards

This tool is implemented in Python[53] using a combination of calls to the NuSMV[24] prover and a custom library called PySMV (described in the appendix) that allows direct manipulations of state sets in the symbolic automata model.

```
VAR f#ev : boolean ;
ASSIGN init(f#ev) := FALSE ;
DEFINE f#ev_next := (a#ev) | (b#ev) ;
ASSIGN next(f#ev) := a$ev_next ;
DEFINE f#ev_next_because#a := (a#ev) ;
DEFINE f#ev_next_because#b := (b#ev) ;
DEFINE f#violation1 := FALSE ;
DEFINE f#violation := f#violation1 ;
```



Algorithm 1: SMV representation of a pulse OR gate

6.1 Tool Input

The circuit netlist is read using a custom JSON format. Gates are specified using a cube representation of their pull-down network. The circuit description is hierarchical on disk, so on loading it is flatted. The phrases are provided expressed in a user-provided python script, and evaluated at runtime with helper functions provided to describe the phrases. Having both the circuit and the phrases, the regions can be derived by flood fill as discussed in chapter 4.

6.2 Unit Time Model Details

6.2.1 Gate Models

Pulse signals in the SMV model are one variable, the event signal. Data signals become two variables, whether an transition is occurring (called an event in the model for symmetry, even though its not a event in the theoretical sense). For all signals a `sig#ev` state bit encodes whether an action is occurring or not, and for data signals a `sig#val` state bit encodes the value of the gate.

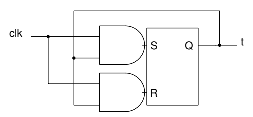
A unit time model of a pulse gate OR gate, represented in NuSMV syntax is shown in algorithm 1. By definition of the unit time model, this produces an event in the next state

```
--- Gate t ---
VAR t#ev : boolean ;
VAR t#val : boolean ;
ASSIGN init(t#ev):= FASLE ;
DEFINE t#set := (clk#ev & !t#val) ;
DEFINE t#reset := (clk#ev & t#val) ;
DEFINE t#ev_next := !t#val & t#set | t#val & t#reset ;
ASSIGN next(t#ev) := t#ev_next ;

ASSIGN next(t#val) := case
t#set: TRUE ;
t#reset: FALSE ;
TRUE: t#val ;
esac ;

DEFINE t#violation1  = t#set & t#reset ;
DEFINE t#violation  = (clk#ev & t #ev) | t#violation1 ;


DEFINE t#set_because#clk := (clk#ev & !t#val) ;
DEFINE t#reset_because#clk := (clk#ev & t#val) ;
DEFINE t#ev_next_because#clk := !t#val & t#set_because#clk
| t#val & t#reset_because#clk ;
```



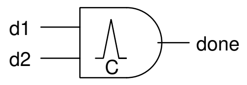
Algorithm 2: SMV representation of a pulse toggle latch

if either of its inputs The `f#ev_next` define is a direct implementation of the production rules for the gate input. The assign statement then uses this value to define the actual next state of the state machine. This define is used rather than a direct approach so that its value can be used later in computing the fast-slow labelling.

Additionally the "next because" defines show if that particular input could be responsible for the signal firing, for tracing purposes. Note that defines like this do not represent overhead in extra variables or extra terms in the transition relation.

A pulse toggle gate is shown in algorithm 2. The set and reset production rules are computed as above. A case statement defines the next value. Whether or not an action occurs is based on the set and reset guard and the current value. A violation is detected

```
VAR done#ev : boolean ;
VAR done#arm1 : boolean ;
VAR done#arm2 : boolean ;
ASSIGN init(done#ev) := FALSE ;
DEFINE done#ev_pdn1 := (d1#ev) ;
DEFINE done#ev_pdn2 := (d2#ev) ;
DEFINE done#ev_next := (done#arm1 | done#ev_pdn1)
    & (done#arm2 | done#ev_pdn2) ;
DEFINE done#arm1_next := (done#arm1 | done#ev_pdn1)
    & ! done#ev_next ;
DEFINE done#arm2_next := (done#arm2 | done#ev_pdn2)
    & ! done#ev_next ;
ASSIGN next(done#ev) := done#ev_next ;
ASSIGN next(done#arm1) := done#arm1_next ;
ASSIGN next(done#arm2) := done#arm2_next ;
DEFINE done#violation := FALSE ;
-- because terms trimmed
```



Algorithm 3: SMV model of a consensus gate

```
-- Basic gate model as above
VAR f#ev_fast : boolean ;
DEFINE f#ev_fast_next := (a#ev & a#ev_fast) | (b#ev & b#ev_fast) ;
ASSIGN next(f#ev_fast) := (meta#apply_speed_cur & f#ev_next) ?
    f#ev_fast_next : (TRUE union FALSE) ;
END
```

Algorithm 4: SMV timed extensions to a pulse OR gate

if the inputs try to set and reset the gate at the same time. The “next because” defines are more involved for a data gate, due to multiple sets of production rules.

Algorithm 3 shows the model of a pulse consensus gate. In addition to the output state bit, the gate has internal state representing which of its two pull down networks has been activated in a previous state and is waiting for the other PDN to pull down before the gate produces its output.

6.2.2 Speed Models of Gates

Algorithm 4 shows the extra details added to a pulse gate to add the fast-slow labelling. This is based on the definition of the labelling that an action is labelled fast if it would have been

The guard define is similar to the original function, but with the appearances of the original event limited to only count those events if they were labelled fast. This models the definition that a action is labelled fast if its occurrence would have been sponsored only considering the fast input events.

Whenever fast slow labelling is being applied, it is applied on either the first or second step of a trace. `meta#apply_speed_cur` is a global bit which represents if labelling is being applied, it becomes true on the step where labelling is applied throughout. Labels are applied and propagated on the first step of the trace for coherence depth determination. They are applied on the second step of for path elimination by coherence, as will be discussed below.

For phrase inputs, the phrase state machine is represented as one bit per state. A one-hot invariant is applied, to ensure only one state is generated. The input signals are initially declared as free variables and then constrained by invariants of the form ”if in this input state, these actions occur”. For the output phrase recognisers, two bits are used. A state possible bit is set by the transition relation based on the states recognised in the previous step. A second bit then indicated if the state is actually recognised, based on this possible bit and the signals observed in the system.

The speed labels on the phrase inputs are generated based on a thermometer code counter that counts off the number of states from the start of labelling up to the input coherence depth. In this window input signals take arbitrary labels. After that all input signals have the same label taken from a global state bit which is set non-deterministically at the start of simulation.

6.2.3 Model Versions

Using these constructions several variations of the automata model are constructed, each containing variations and a subset of the features described to make the suited for particular verification operations. The starting state of the SMV model can be either the true initial state of the unit time model at the start of the input phrase, or after the first model has been built this way a initial state of all reachable states can be used. This is produced using the reachability analysis that can be performed using NuSMV. Computing on this model gives an implicit LTL eventually operator around any proposition, which is useful for bounding the required depth in bounded model checking. The models constructed are:

- A "basic" model contains no speed information and its initial state is the start of input.
- A "tracing" model contains doesn't contain speed information, and its initial state is the reachable set. This also omits the output recogniser for execution speed.
- A "speed" model contains speed information and its initial state is the reachable set.

6.3 Static Checks

Having constructed the model, two properties can be checked immediately without reference to the fast-slow labelling. One is that no gate violations occur. Gate violations are where the inputs of a gate in a given state are invalid. For all gates, it must be ensured that for any pulse in a pull down network, none of the data signals guarding that pulse change at the same time as it. For pulse gates this is the only hazard. For data gates, the case where both the set and reset inputs occur simultaneously must also be handled.

In model construction, first a `sig#violation1` is constructed describing the gate type specific violations (i.e. simultaneous set and reset). Then a `sig#violation` is constructed from the sum of that and the input pairs hazard that occurs for all gates. A global "no violations" define is made as the negation of the sum of the gate violation bits, and this is checked as an invariant in NuSMV.

Currently Pulse consensus gates have no type violation behaviour. One behaviour this model includes is that the consensus gate silently absorb a second pules on the same input before the other input arrives. This is allowed since that behaviour is sometimes intended but this could be added as a violation if it were not intended.

The other check is that output phrases are correctly recognised. Two properties are checked. One is that the recogniser machine always remains alive for all states. This is checked as an invariant in NuSMV. Then it is checked that the output recogniser either reaches a terminal state, for when that phrase is being produced in that execution, or remains in the initial state for when that phrase is not produced in that trace. Since regions may have multiple output states, these properties are checked for all outputs.

6.4 Coherence Depth Determination

If a model has coherence depth D then by the Dth step of any trace all signals will have the same label. e.g. to check if the system has coherence depth 3, this can be established by proving that no traces satisfy the CTL formula:

```
DEFINE meta#signals_coherent_fast := (!sig1#ev | sig1#ev_fast)
    & (!sig2#ev | sig2#ev_fast) & ... ;
DEFINE meta#signals_coherent_slow := (!sig1#ev | !sig1#ev_fast)
    & (!sig2\#ev |! sig2#ev_fast) & ... ;
DEFINE meta#input_coherent_fast := meta#input_done
    | (meta#input_coherent & meta#input_fast) ;
DEFINE meta#coherent_fast := meta#signals_coherent_fast
    & meta#input_coherent_fast ;
-- Similar defines for slow coherence --
DEFINE meta#is_coherent := meta#coherent_fast | meta#coherent_slow
```

Algorithm 5: Defines for determining slow coherence

$$(!meta\#apply\_speed\_cur)|\,(EXEXEXmeta\#is\_coherent)$$

Where `meta#is_coherent` is defined as requiring all the signals to have settled to the same label. Where the input is still ongoing, the input must have settled to coherence itself, and that label must be the same as the signals within the circuit. Otherwise while the circuit gates may presently have the same label the input may reintroduce the other label later. By this definition, it is not needed to explicit specify that the model remains coherent once it is, since by the definition of the labelling functions, it remains true once set true.

This is checked against the ”speed” model described above, further constrained so that the labels are applied from the initial state. This is checked with NuSMV’s BDD based CTL checker, which was found to perform better than bounded model checking in practice.

Given a region, its coherence depth D can be established given its input coherence depth D_{in} by building the ’speed’ model with an input generator with the correct depth. Then we apply the above test for increasing depths D until a correct value is found. This search definitely terminates since the behaviour within a region is required to be terminating. Having established the coherence depth for the whole region, output coherence depths for each output phrase can be established by testing that the output signals have settled to the same label as the circuit eventually settles on. For example, to see if an output phrase has coherence depth 2 in a region that has coherence depth 3 the following formula is disproved:

$$\begin{aligned}&(!meta\#apply\_speed\_cur)|\\&\quad(EXEX(meta\#output0\#coherent\_fast\&EXmeta\#coherent\_fast))\,|\\&\quad(EXEX(meta\#output0\#coherent\_slow\&EXmeta\#coherent\_slow)\end{aligned}$$

Note that it is sufficient to only check the output is coherent on that one step and not re-check it on all later steps. If a counter-example exists where the output becomes coherent on the required step and then changes on a later one, the given formula will flag that counter-example just from a different initial position of the trace.

Now that a procedure for determining one regions coherence depth has been described, this can be used to obtain the actual coherence depths for each phrase and region found in a system by iteration to an upper fixed point. It is initially assumed that each phrase has coherence depth 1 unless an explicitly higher number has been given in the input. Coherence depths for each region and output phrase are then computed, and the phrase coherence depths are increased if they are generated as output with a larger coherence depth than was previously seen. This is repeated until a fixed point is reached.

6.5 Path Tracing

6.5.1 Hazard Generation

An initial set of hazards that might occur in a region can be identified by inspection of the circuitry. Specifically we look at each gate and take pairs of incoming signals. The typing of the signals determines what type of hazards can occur. A hazard is identified by this pair of signals, the nature of hazards, and as a consequence of the nature of whether the signal in the slow path occurs in the same timestep as the fast one or as an earlier one.

Timing constraints back-propagated from a later region also have information on the states of the phrase that the events occur in, and how many timesteps are between them.

6.5.2 Path Tracing

The paths through the circuit that lead to a particular hazard can be found by a search backwards in time from the final state where the hazard itself is. A node in the search tree has the following information:

- A pointer to the hazard identity described above.
- The two current signals for this step in the trace.
- A set of SMV states where possible for that location in the trace.
- A pointer to the next point in the trace forward in time.
- The depth of the node in the tree.

The SMV state sets from this are implemented using the 'tracing' model described above. The signals can also be two atoms, '@*past*' meaning the tracing has not reached back to signal in past in the shorter trace, and '@*input*' meaning this signal has been traced back into the input region and tracing is continuing waiting for the other side of the trace to cross the input. Since the tracing search happens backwards in time, the next node forward in time is the search nodes parent.

To start tracing back a given node, a hazard identity is taken and used to build the starting tracing node. The signals are the two signals of the hazard if simultaneous, or the fast signal and '@*past*' for the slow one if they are not.

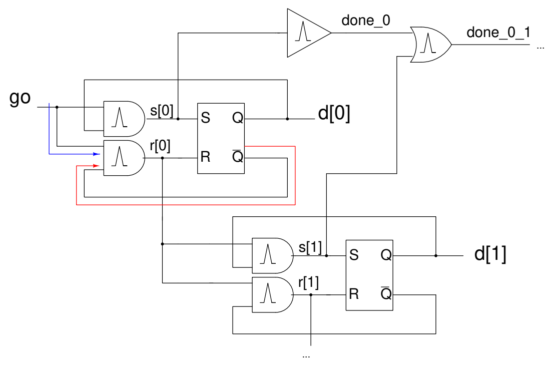
To visit a node in the search tree, looping traces are detected, and if this occurs the states from the state set that correspond to looping are eliminated. It is then checked if the trace is a complete trace, that is both the traced signals are the same or both are inputs. If so this trace is emitted for refinement as discussed below.

If it is not a complete trace, pairs of ancestor signals are generated topologically for the two signals in the current node. "@past' can trace back either to '@past' or to the slow signal from the hazard id. Primary inputs and '@input' trace back to '@input', assuming the other signal of the pair will trace back to a primary input later on. Having generated the possible signal pairs, the current state set is stepped back in time and see if there is a subset of that prior state set where the two candidate signals occur and the data guards are such they can lead to their intended successors. If there is one, a further search tree node is generated to visit later.

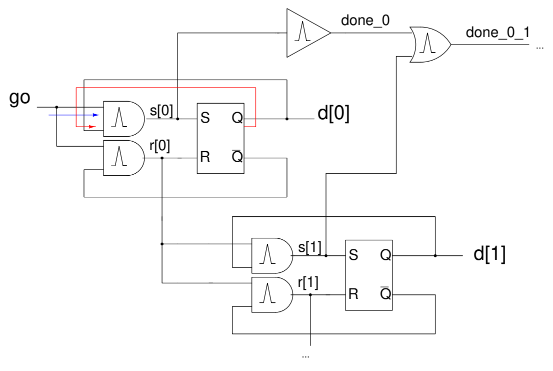
Searching back this search tree in a recursive fashion will lead to a series of candidate traces to be further refined.

6.5.3 Tracing Example

Figures 6.1 through 6.5 show some examples of the path tracing algorithm. First note in figure6.1 that while the same hazard may be identified at multiple gates inputs,

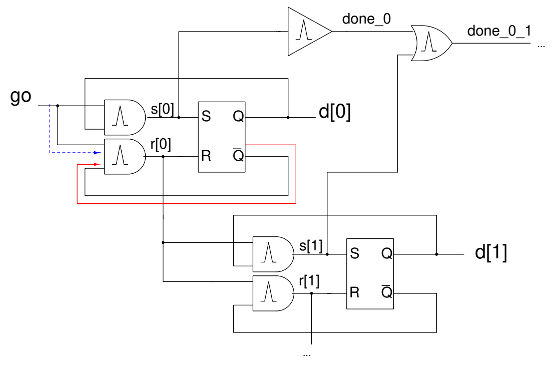


(a)

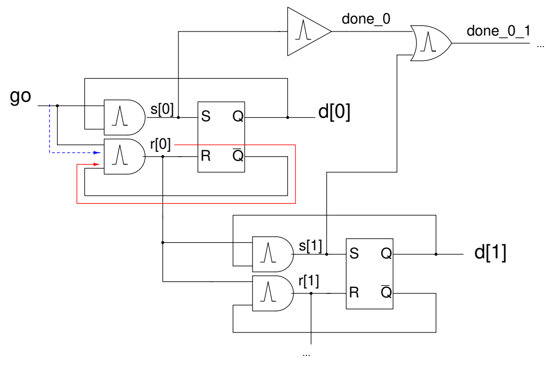


(b)

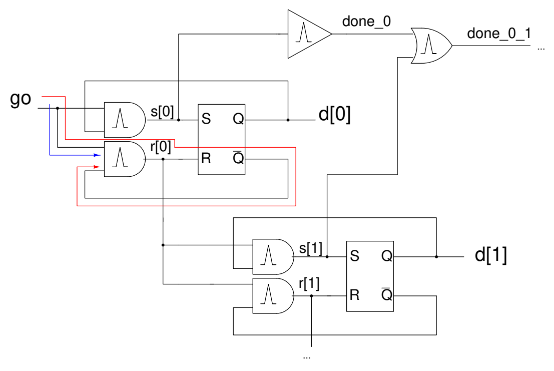
Figure 6.1: Hold hazard between `d[0]` and `go` found at both `r[0]` and `s[0]`



(a)



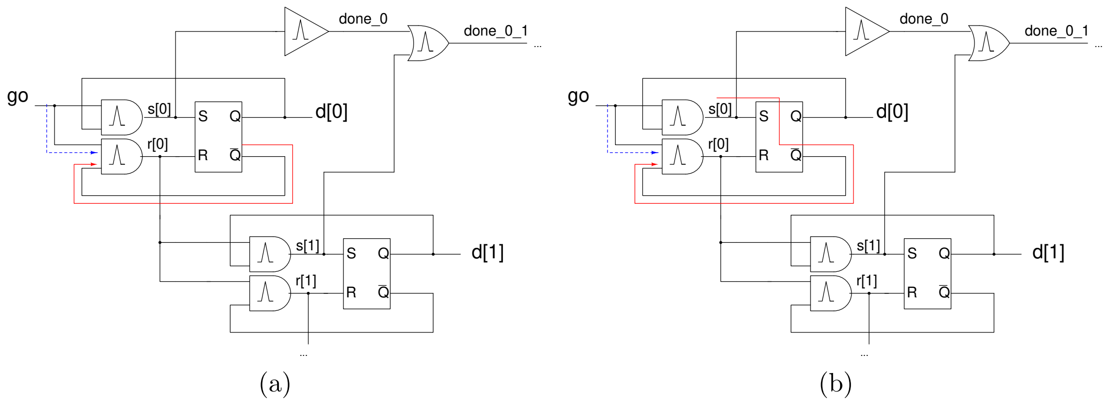
(b)

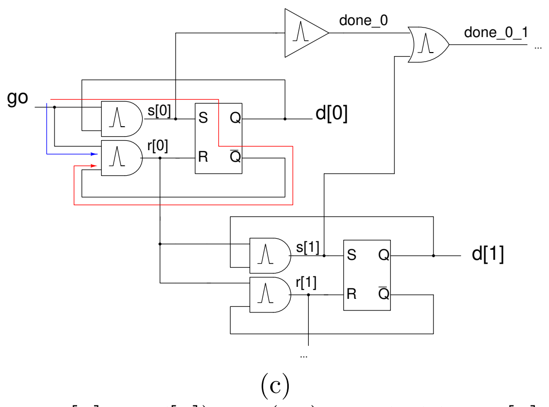


(c)

$$t(go \to r[0] \to d[0]) > t(go) + t\_hold \text{ at } r[0], s[0]$$

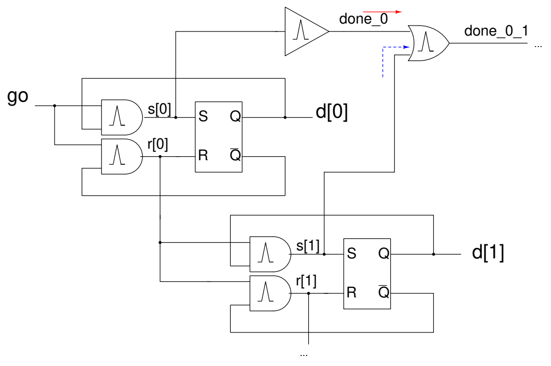
Figure 6.2: Hold hazard traced back to one possible path inequality



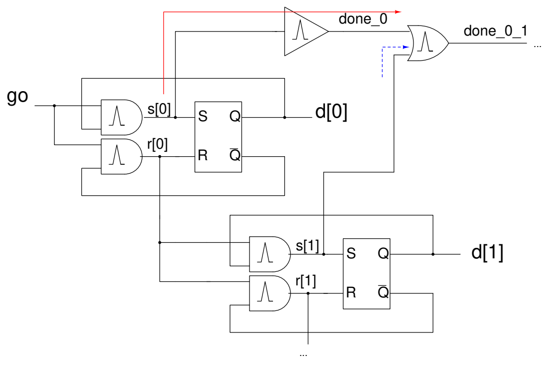


$$t(go \to s[0] \to d[0]) > t(go) + t\_hold \text{ at } r[0], s[0]$$

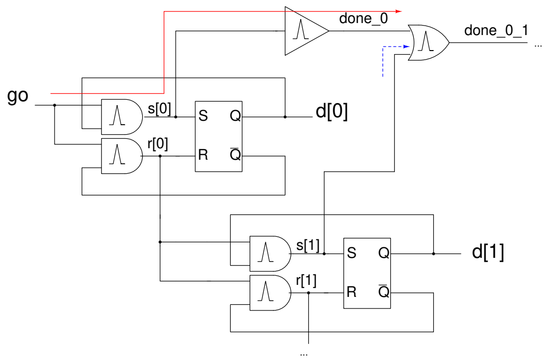
Figure 6.3: The same hold hazard traced back to a different path inequality



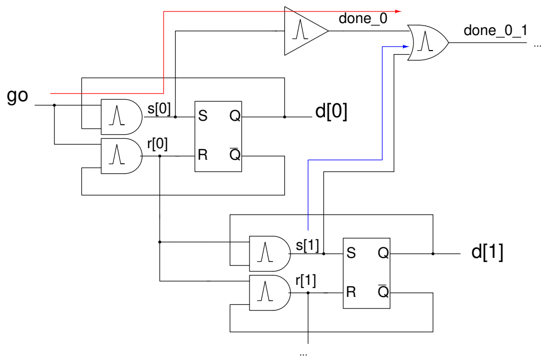
(a)



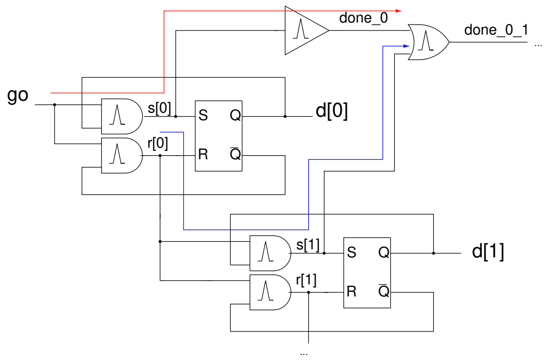
(b)



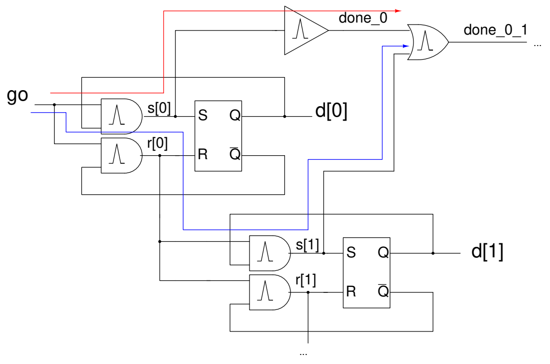
(c)



(d)



(e)



(f)

$t(go..go) + t(go \rightarrow s[0] \rightarrow done\_0) > t(go \rightarrow r[0] \rightarrow s[1]) + t\_retrig$ at $done\_0\_1$

Figure 6.4: A retrigger hazard traced back to the input phrase

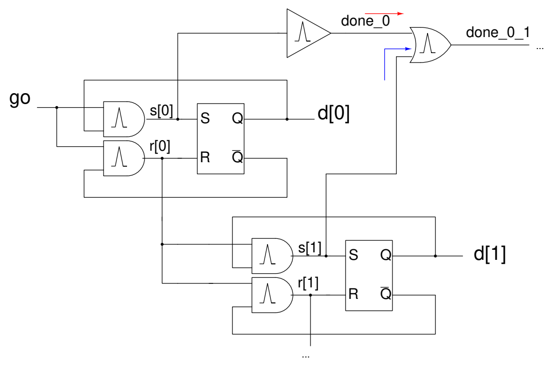


Figure 6.5: Example of a coalescence hazard that is topologically possible but does not occur in the unit time model

the same tracing applies to all of them and only at path checking time must the path be checked for each. Figure 6.2 and 6.3 shows the tracing of that hold hazard back to two possible paths. Figure 6.4 shows the tracing of a retrigger hazard in the completion tree of the counter, which traces back to the input hazard. Figure 6.5 shows a coalescence hazard that is topologically possible (the gate has two pulse inputs) but does not occur in the unit time model (since both of those input pulses never actually occur). It will be generated by the hazard generator but eliminated in the first state of the tracing hazard.

6.5.4 Trace Refinement

Have generated a set of possible traces above from our search tree, the linked list form of the trace tree if first confirmed into a linear form, which we call a hazard path.

For those hazard paths that terminate on primary inputs rather than on the internal nodes, the tracing process established which input signals correspond to those nodes but not which states of the generating machine do. Thus possible states of the input machine on those two timesteps are generated and checked against the state set BDD to see if they are plausible.

Finally all the generated traces are checked with a final linear model check. This is performed using the 'speed' model discussed above. The primary purpose of this is to check if the trace is not in fact eliminated by some synchronisation mechanism and thus only shorter paths (after that synchronisation mechanism) need apply. For this check the labels are applied at the second timestep, not the first, since the first step is the common ancestor before dispersion has built up. This is performed in NuSMV's symbolic checking using a CTL constraint.

For traces which come back from an output phrase, it is also this point that the state information in the output trace is enforced, since the tracing BDD model omits the recogniser for speed.

6.5.5 Tracing Through Regions

Having traced those hazards that occur topologically back to their origins, some of those origins may be on the input of that region. If that input phrase is one that is generated by another region in the circuit rather than a primary input, it is necessary to quantify the timing relation between those two signals. An extra constraint is generated and traced as described above in all the possible predecessor regions which generate the phrase.

6.6 Conclusions

In this chapter the implementation of a computer tool based on the models discussed thus far has been discussed. The implementation of the unit time model, including phrase boundary conditions, into SMV is discussed. It is then shown how this model can be used to check directly checkable violations. Then a coherence depth for the circuit can be measured. Finally a backtracing search "tracing" procedure is described to find the timing inequalities that must be enforced to ensure correct behaviour of the circuit.

Chapter 7

Results

In order to evaluate the methodology and the tool that have been presented, the tool was run on several examples systems and the results are presented here. Further the tool was run on a simple test circuit to identify the hazard in the circuit. The predictions on when the circuit should operate under voltage noise was then compared with spice results of the circuit under voltage noise.

7.1 Systems Results

The 3-bit counter from 2.3 discussed thus far is evaluated as a basic example. Two different varieties of 4-bit pulse ser-des system were designed, one with a tree serialiser (as was shown in figure 2.4 and one with a linear serialiser closer to what was implemented in silicon in [1]. Both use tree deserialisers, as shown in figure 7.1.

A 4 bit ripple carry adder is also tested, implemented using a one-hot design style. A version of the the circuit from figure 4.2 above was also tested, with the iterative circuits being 3-bit counters looped on themselves until they overflow to achieve a delay of approximately 40 cycles.

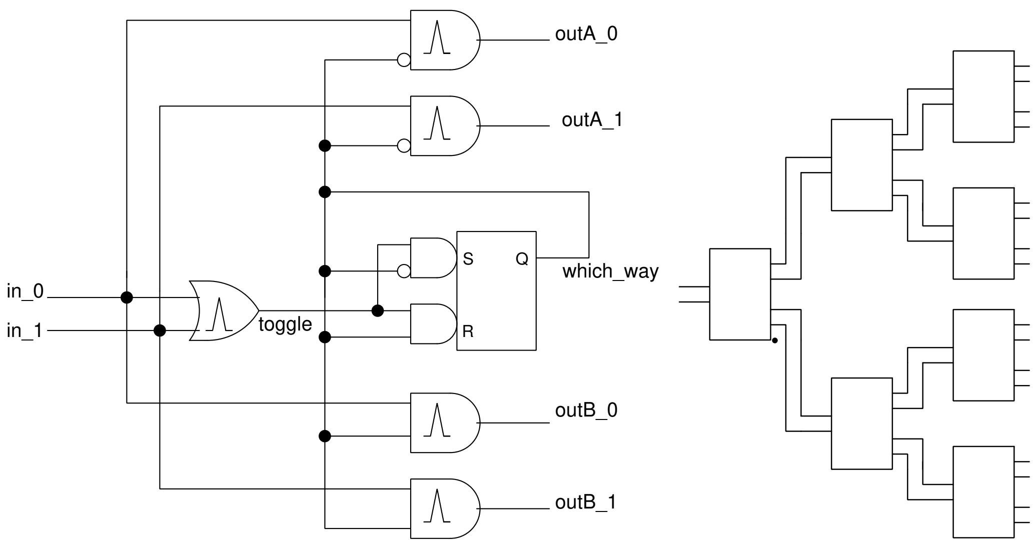


Figure 7.1: Pulse Tree Deserialiser

Circuit	Setup	Hold	SR Ord.	Retrig	Thru
3-bit counter	2	8	2	8	0
4-bit Serialiser (tree)	4	0	0	16	24
4-bit Serialiser (linear)	4	0	0	20	24
4-bit Deserialiser	80	89	40	48	0
4-bit Ripple Carry Adder	241	0	0	0	0

Table 7.1: Details of the constraints found in some regions of the test circuits

Results are presented in two ways, since the tool analyses a whole system but produces constraint databases per region. Table 7.2 shows the sizes and runtimes of the various systems considered. Table 7.1 shows the number of hazards for some key regions of interest in these circuits. Runtimes ranged from 3 sec for the binary counter to 32 sec for the SerDes system with the linear serialiser. These results were obtained on a computer with an Intel i7-4770 CPU at 3.40GHz and 32GB of DDR3 memory.

Circuit	System Size	Largest Region	Time
	(gates)	(gates)	sec
Counter	12	12	3.0
Ser-Des system (linear)	54	31	27.1
Ser-Des system (tree)	52	29	26.5
4-bit ripple carry adder	28	28	24.1
Region join test	39	18	8.7

Table 7.2: Run-time results for various systems

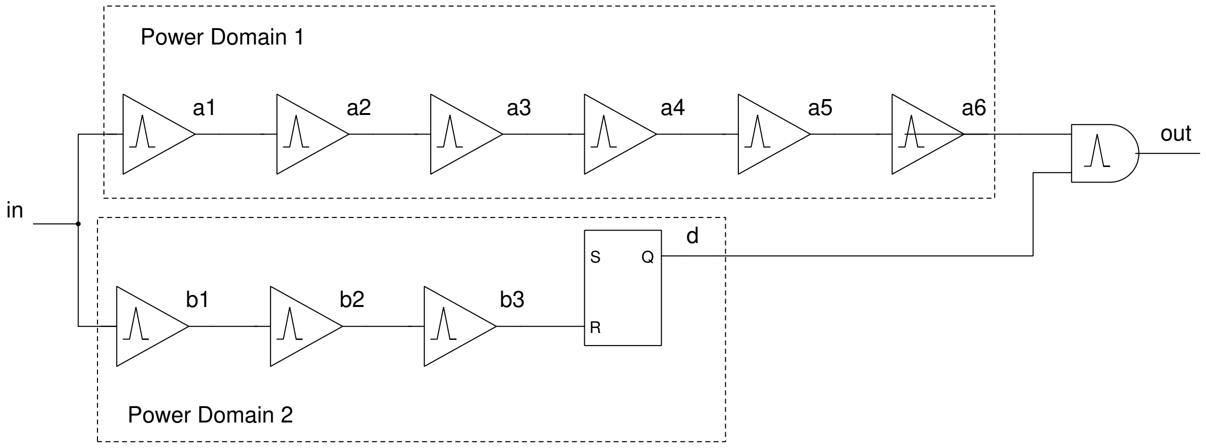


Figure 7.2: Circuit for SPICE Comparison Test

7.2 Spice Comparison

To compare the timing constraints produced by the tool directly against actual circuit behaviour, the circuit in figure 7.2 was constructed both in the tool presented and in SPICE. It was based a pulse gate set in a 130nm process[1]. Given the available cell set, the buffers were implemented as OR gates with one of the inputs tied to ground. In the intended behaviour of this circuit, data signal `d` falls before pulse `a5` and thus `out` is not produced. Voltage noise is applied, consisting of 20% deviation around a 1.5V nominal voltage, for a range of 1.2V to 1.8V. This voltage noise is applied symmetrically so that a gate running at 1.2V had its ground at 0.15V and its supply is a 1.35V. These variations are applied to the circuit in two power domains. The input pulse and the `out` gate are held at the nominal 1.5V.

As designed, tracing the circuit in the tracing tool yields one hazard: $t(in \rightarrow a1 \rightarrow a2 \rightarrow a3 \rightarrow a4 \rightarrow a5 \rightarrow a6) > t(in \rightarrow b1 \rightarrow b2 \rightarrow d3 \rightarrow d) + t\_hold$ at $r[0], s[0]$ Incorrect behaviour occurs when the voltage in power domain 1 gets higher, causing the intended later event `a5` to occur earlier, or equivalently with the voltage in power domain 2 becomes lower and `d` changes later. The latch used happens to have a much longer propagation delay than the pulse gates, hence the skewed hazard.

Next the gates used were characterised. This was achieved placing each gate in a separate test jig in SPICE. The main parameter characterised for was propagation delay. A summary of data is shown in table 7.3, though the actual characterisation is based on 7 voltage points in 0.1V intervals. For pulse signals, the propagation times is based on when the pulse rises through 70% of nominal supply voltage. This was chosen based on this 70% point at the input corresponding to the start of internal switching in the critical node of the gate. This minimises the effect of slope on the results, otherwise a multi-dimensional characterisation would be needed. For data signals a 50% switching

Voltage	1.2 V	1.5 V	1.8 V
PulseOrHH	36.2 ps	29.3 ps	25.2 ps
PulseAndHH	38.9 ps	30.2 ps	25.7 ps
PulseLatch	123.9 ps	103.9 ps	94.8ps

Table 7.3: Summary of Characterisation data for gates used

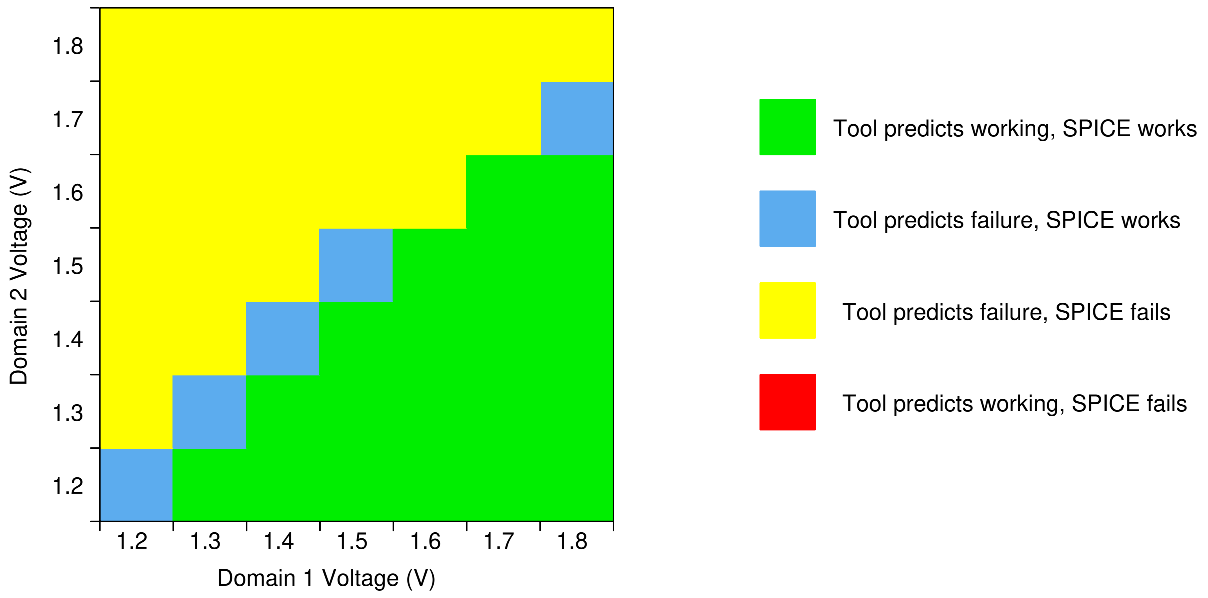


Figure 7.3: Circuit function for different conditions, as predicted by this tool and measured by SPICE

point is used.

The setup time for falling data inputs of the PulseAndHH gate was also measured, and found to be -10ps. That is, the correct behaviour (no output) is achieved even when the data signal changes 10ps after the pulse input. For this characterisation test to give a strict bound, correct behaviour is defined as the output not rising above 0.15V (10% of nominal supply voltage). This characterisation is strict for the reasons discussed in chapter 2.

The behaviour of the circuit was then simulated in SPICE sweeping both power domains independently from 1.2V to 1.8V in 0.1V steps. Correct behaviour here is determined by the more liberal definition of placing an extra gate in series with the output and ensuring that `out` does not rise sufficiently to trigger any behaviour from that gate, as this is what would matter in practice. A script was also written to load the inequalities produced by the timing tool and the characterisation data and substitute appropriately to predict whether the circuit should work under those conditions. The result is shown in figure 7.3. It can be seen that the tool broadly agrees with the SPICE predictions, and where it differs it is more conservative, predicting the circuit will fail when it actually succeeds in SPICE. This difference is possibly due to errors in the model (the computed propagation times on the path differ from spice measurements by up to 3%) or due to the conservative hazard time characterisations.

7.3 Conclusions

In this chapter, the results of applying the models and tools thus far in this work have been discussed. The tool has been applied to several medium-sized example systems producing comprehensive databases of timing constraints for those. The tools results have also been compared with SPICE for a small circuit. Even with the simplified characterisation performed here good agreement is achieved between tool predictions and actual SPICE.

Chapter 8

Conclusions

In this theis the objective is to design models for asynchronous circuits to be:

- Sufficiently general to include hand-designed pulse gate circuits.
- Amenable to automatic timing verification.
- If possible, encompass known pragmatic techniques for asynchronous closure.

To this end, two models have been presented for timing verification of pulse gate systems. First the "unit time" model has been presented to capture local behaviour making strong timing assumptions and then the "phrase" model has been introduced to model system behaviour and also describe the communication between the unit timed regions.

A notion of "coherence depth" to measure how much dispersion can build up in a circuit has been introduced, and it has been proven how this property can limit the amount of timing checks that need to be done.

A computer tool to verify against the timing dispersion that does build up has been presented, and it achieves this by ensuring that the coherence depth of each region of the circuit is finite and then producing timing path inequalities. This allows the tool to be run once before layout and this timing inequality database to be used to inform placement and routing. Results from this computer tool have been discussed.

The system level implications of the phrase model have been discussed, how they enable a top level view of the circuit to be taken.

8.1 Completeness of Models

Here the assumptions that are made by timing models in this thesis, and how they have been addressed.

- To show dispersion is finite, coherence depth has been defined and determination of coherence depth by computer has been implemented.
- Preventing static violations of gate behaviour has been implemented in through gate violation invariants in the unit time model.
- Verification of the timing hazards that do occur has been implemented in the tool through tracing of timing path inequalities, which can then be substituted into with values from a timing library post route.
- Proving that phrases describe the possible communications that occur in the system has been implemented by dividing up the system into regions based on phrases, and verifying that the output phrases do contain the regions outputs with language containment.
- Proving the non-interference between regions by an outer speed implemented has been discussed in theory but not yet implemented. This also requires some extensions to be added to the unit time modelling part of the tool, in particular precondition case analysis and determination of reached signals.

- One final issue that would have to be addressed with that speed independent model is the fact that phrases are not atomic actions, and thus could proceed in an overlapping manner. This does not present an issue for immediate predecessor-successor regions since the phrase model describes all interaction, but the second next successor if it assumed phrases were atomic may try to re-use signals from the first region while that had not yet quiesced. Parallel version of this are not an issue due to the speed independent assumption of the model, all overlaps will be reached by this.

8.2 Future Work

The most immediate future work is scaling the tool presented up to larger examples circuits. This should be relatively straightforward as earlier versions of the tool had much better performance before the move to the SMV based version (which was necessary to implement the current phrase model).

The implementation of a tool to perform the presented speed-independent system level theory would be the next step towards a tool suite providing complete verification coverage of pulse gate systems.

One omission from the model that was not obvious from considering hand designed circuits was that of an arbiter. However most theory on asynchronous system completeness assume some sort of arbiter, if not directly then as a mutual exclusion block. Adding this would allow completeness proofs such as [41] to apply to this model.

Appendix A

PySMV

PySMV is a library for dealing with SMV models of automata directly within a program and directly manipulating sets of states. For the automata models used in this work, it was desirable to construct them in a fashion that would allow both analysis in existing state machine analysis tools and also allow the software being developed to directly manage state information. The NuSMV solver was chosen to prove pre-packaged analysis. A compiler for a subset of its input language was then implemented to allow the same model with direct access to sets of states. The SMV language was chosen due to its wide use in different automata verifiers, the fact that it directly implements synchronous logic(as opposed to say Verilog which is an event simulator language that people happen to write synchronous logic in).

Specifically, a subset of the language was implemented. The only variable types allowed are individual boolean variables. None of the integer types or bit vectors or arrays are supported. These were not implemented since they were not needed for the models in this work, but could be readily added if needed.

PySMV is a library for the PythonPython[53] programing language. A BDD representation of the sets of states is used, implemented using CUDD[27] through the

PyCUDD[54] wrapper. PyCUDD as used in this project has been adapted from the published Python 2 version to work with Python 3.

THe PySMV library exports two entry points, `pysmv.read_file` or `pysmv.read_str` which take a SMV model, bt filename or text respectively and compile this to a `SMVAbstrctModel` object. This object contains a abstract syntax tree that has had all necessary backend-independent processing it. The compilation can then be finished by calling that objects `to_bdd()` method which compiles that AST into a BDD model `SmvStateSetModel_BDD`. This architecture was designed to allow potential future other backends but none were implemented as these analysis were performed with NuSMV.

Once a BDD model has been constructed, a starting set of states can be obtained from the model, either the initial states or reachable states. This `SMVStateSet_BDD` object can then be manipulated with usual set operations, a subset that satisfies a certain predicate constraint take, or temporal operations (backwards and forwards sets and the CTL exists-eventually operators)

A.1 AST Compilation

The SMV is first tokenised and parsed using a lexer and parser implemented with the PLY (Python Lex-Yacc) library [55]. As its name suggests this is python version of the traditional UNIX Lexx lexer generator and YACC parser generator. It first performs regular expression based tokenisations and then a LALR grammar parsing. From this an initial abstract syntax tree is produced.

After this the syntax tree is processed with a series of visitor-pattern passes. First, if the design has multiple modules it is flattened. Modules are topologically sorted using the toposort library, then for each module in that topological sort order it is examined and the submodules calls in that module are expanded into the parent module, with the expressions being copied over and variable names in them either substituted if they are arguments or else prefixed with the parent modules instance name of the submodule, plus a dot.

Next the statements of SMV module are separated into their categories, since these need to be treated differently in the BDD compilation:

- `INIT` statements specify formulae that are true in the initial state.
- `TRAN` statements give parts of the transition relation of the system.
- `INVAR` statements specify invariant formula that hold throughout the model.
- `ASSIGN` statements can be any of the above three depending on their form of their left-hand side, in all cases defining a specific variable in that context
  - A LHS of `init(var)` gives the initial value of a variable
  - A LHS of `next(var)` gives the next value of a variable in the transition relation.
  - A LHS of `var` gives that that variable is invariantly equal to the the given formula.
- `DEFINE` statements give a name to a sub-formula to allow its re-use in multiple statements.

A name checking pass ensures names are not redefined.

Since defines are substituted into other formulas, and define statements can potentially by defined in terms of other defines. Thus an order is found whereby defines that depend on other defines are compiled later. A “sort defines” pass produces this evaluation order for the define statements in the model using a topological sort. If defines are cyclic an error is thrown.

Finally one particular feature of the NuSMV syntax is that in general transition relations, the `next()` function can wrap arbitrarily formula indicating that formula is true in the next state rather than the current state of the transition relation. Since this operation distributes over other boolean operations this pass finds calls to “pushes” `next` to the variable leaves of the parse tree.

A.2 BDD Model Compilation

Having a abstract syntax tree, the next step is to compile a BDD model. The first step in compiling a BDD design is to allocate BDD variable numbers for the variables in the design. Two bits are allocated for each variable, one for the current state value and one for the next state version in transition relations. No variable ordering is currently performed in PySMV, the variables are allocated in the order presented in the model file. Ordering can be applied when generating the model file as necessary. Four ancillary variables `aux1`to `aux4` are also allocated for use in intermediate products. These variables are all dereferenced once in a throw-away fashion to ensure they are pre-allocated, since CUDD does not allow the permute operation to make new variables with higher indexes than it has seen before. Since the number of variables in the model is now known

Then statements are evaluated, first the define statements are evaluated and the results stored for re-used. For each define the result is stored for future re-used. Then initial, transition and invariant statements are compiled and initial, transition relation and invariant BDDs are made as a product of this. Statement evaluation is performed recursively. The intermediate value at each step is not the value itself, but rather the relationship `aux1` $\Leftrightarrow$ `value`. This is done to facilitate non-determinism with SMVs `union` operator. When a variable or constant is referenced, this formulation is directly applied to give the intermediate value. To implement a deterministic operation *op* the inputs are relabelled into `aux2`, `aux3`, etc. These and also the relationship $\texttt{aux1} \Leftrightarrow \texttt{aux2}op\texttt{aux3}$ are producted together. `aux2` and `aux3` are then existentially abstracted out of this result. The union operator is implemented similarly, with the relationship is $(\texttt{aux1} \equiv \texttt{aux2}) \vee (\texttt{aux1} \equiv \texttt{aux2})$ used instead. This construction is from the original SMV[23].

For defines, the resulting formula is stored in this `aux1` relationship form, since it is intended to be referenced in the AST like this. For other formulas we are only interested in the truth of the formula so they are anded with `aux1` and then that is existentially abstracted out to give the bare proposition.

A.3 Model Manipulation

State sets from the model can then be manipulated using opaque objects. The initial state is based on the product of the initial constraints and invariants in the model. The reachable states are produced using the obvious step-forward and union fixed point algorithm. The first time this is requested it is computer and cached in the model so subsequent requests do not recompute it.

The state set objects support a `constraint` method, which takes a subset of that state where a given variable has a given value. Constrains can also be applied to require a d. The binary set operations of intersection, union and difference are also supported. A temporal step forward operation is supported. This is implemented in the usual symbolic way of ANDing with the transition relation, existentially abstracting the current state, permuting the next state to the current state and then ANDing invariants onto that result. A temporal step backwards is similarly implemented.

Lastly a method is provided to compute the CTL operator EF (exists eventually) on the given set of states. This computes the set of states that have a path to eventually reach one of the states in the input set. This is implemented using the standard step backwards and union fixed point algorithm.

Appendix B

SMV Model Example

In this appendix we present the complete source code of a model. Specifically the circuit netlist and the phrase defintion are included as are the NuSMV models produced by the tool presented for this. Specifically the basic and "speed" models are presented, the tracing model is a subset of the basic model. The circuit preseneted is the binary counter from figure 2.3

B.1 Circuit Netlist

```
{"modules": [ {
        "name": "main",
        "inputs": ["clk"],
        "outputs": ["clk_out", "d0", "d1", "d2"],
        "events": ["clk", "s0", "s1", "s2", "r0", "r1", "r2", "done_0",
            "done_0_1", "clk_out"],
        "data": ["d0", "d1", "d2"],



        "pgates": {
                "s0" : ["clk !d0"],
                "r0" : ["clk d0"],
                "s1" : ["r0 !d1"],
                "r1" : ["r0 d1"],
                "s2" : ["r1 !d2"],
                "r2" : ["r1 d2"],
                "done_0": ["s0"],
                "done_0_1": ["done_0", "s1"],
                "clk_out": ["done_0_1", "s2", "r2"]

        },
        "dgates": {
                "d0": ["s s0", "r r0"],
                "d1": ["s s1", "r r1"],
                "d2": ["s s2", "r r2"]
        }}

], "phrase_file": "binary_counter_phrases.py"}
```

B.2 Phrases

```
clk_sig = sig('clk')

input_state = state().sig(clk_sig)
input_state.wait(5).sig(clk_sig).wait(5).done()
input_phrase = phrase().start(input_state)
```

B.3 Basic Model

```
MODULE main
```

```
--- Gate s0 ---
DEFINE s0#violation := (clk#ev&d0#ev)|s0#violation1 ;
VAR s0#ev : boolean ;
ASSIGN init(s0#ev) := FALSE ;
DEFINE s0#ev_next := (clk#ev & !d0#val) ;
ASSIGN next(s0#ev) := s0#ev_next ;
DEFINE s0#violation1 := FALSE ;
DEFINE s0#ev_next_because#clk := (clk#ev & !d0#val) ;

--- Gate r0 ---
DEFINE r0#violation := (clk#ev&d0#ev)|r0#violation1 ;
VAR r0#ev : boolean ;
ASSIGN init(r0#ev) := FALSE ;
DEFINE r0#ev_next := (clk#ev & d0#val) ;
ASSIGN next(r0#ev) := r0#ev_next ;
DEFINE r0#violation1 := FALSE ;
DEFINE r0#ev_next_because#clk := (clk#ev & d0#val) ;

--- Gate s1 ---
DEFINE s1#violation := (r0#ev&d1#ev)|s1#violation1 ;
VAR s1#ev : boolean ;
ASSIGN init(s1#ev) := FALSE ;
DEFINE s1#ev_next := (r0#ev & !d1#val) ;
ASSIGN next(s1#ev) := s1#ev_next ;
DEFINE s1#violation1 := FALSE ;
DEFINE s1#ev_next_because#r0 := (r0#ev & !d1#val) ;

--- Gate r1 ---
DEFINE r1#violation := (r0#ev&d1#ev)|r1#violation1 ;
VAR r1#ev : boolean ;
ASSIGN init(r1#ev) := FALSE ;
```

```
DEFINE r1#ev_next := (r0#ev & d1#val) ;
ASSIGN next(r1#ev) := r1#ev_next ;
DEFINE r1#violation1 := FALSE ;
DEFINE r1#ev_next_because#r0 := (r0#ev & d1#val) ;

--- Gate done_0 ---
DEFINE done_0#violation := done_0#violation1 ;
VAR done_0#ev : boolean ;
ASSIGN init(done_0#ev) := FALSE ;
DEFINE done_0#ev_next := (s0#ev) ;
ASSIGN next(done_0#ev) := done_0#ev_next ;
DEFINE done_0#violation1 := FALSE ;
DEFINE done_0#ev_next_because#s0 := (s0#ev) ;

--- Gate d0 ---
DEFINE d0#violation := d0#violation1 ;
VAR d0#val : boolean ;
VAR d0#ev : boolean ;
ASSIGN init(d0#ev):= FALSE ;
DEFINE d0#set := (s0#ev) ;
DEFINE d0#reset := (r0#ev) ;
DEFINE d0#ev_next := !d0#val & d0#set | d0#val & d0#reset ;
ASSIGN next(d0#ev) := d0#ev_next ;
ASSIGN next(d0#val) := case
                d0#set: TRUE ;
                d0#reset: FALSE ;
                TRUE: d0#val ;
         esac ;
DEFINE d0#violation1 := d0#set & d0#reset ;
DEFINE d0#set_because#s0 := (s0#ev) ;
DEFINE d0#reset_because#s0 := FALSE ;
DEFINE d0#ev_next_because#s0 := !d0#val & d0#set_because#s0 | d0#val &
    d0#reset_because#s0 ;
DEFINE d0#set_because#r0 := FALSE ;
DEFINE d0#reset_because#r0 := (r0#ev) ;
DEFINE d0#ev_next_because#r0 := !d0#val & d0#set_because#r0 | d0#val &
    d0#reset_because#r0 ;

--- Gate s2 ---
DEFINE s2#violation := (r1#ev&d2#ev)|s2#violation1 ;
VAR s2#ev : boolean ;
ASSIGN init(s2#ev) := FALSE ;
DEFINE s2#ev_next := (r1#ev & !d2#val) ;
ASSIGN next(s2#ev) := s2#ev_next ;
DEFINE s2#violation1 := FALSE ;
DEFINE s2#ev_next_because#r1 := (r1#ev & !d2#val) ;

--- Gate r2 ---
DEFINE r2#violation := (r1#ev&d2#ev)|r2#violation1 ;
VAR r2#ev : boolean ;
ASSIGN init(r2#ev) := FALSE ;
DEFINE r2#ev_next := (r1#ev & d2#val) ;
ASSIGN next(r2#ev) := r2#ev_next ;
DEFINE r2#violation1 := FALSE ;
DEFINE r2#ev_next_because#r1 := (r1#ev & d2#val) ;

--- Gate done_0_1 ---
DEFINE done_0_1#violation := done_0_1#violation1 ;
VAR done_0_1#ev : boolean ;
ASSIGN init(done_0_1#ev) := FALSE ;
DEFINE done_0_1#ev_next := (done_0#ev) | (s1#ev) ;
ASSIGN next(done_0_1#ev) := done_0_1#ev_next ;
```

```
DEFINE done_0_1#violation1 := FALSE ;
DEFINE done_0_1#ev_next_because#done_0 := (done_0#ev) ;
DEFINE done_0_1#ev_next_because#s1 := (s1#ev) ;

--- Gate d1 ---
DEFINE d1#violation := d1#violation1 ;
VAR d1#val : boolean ;
VAR d1#ev : boolean ;
ASSIGN init(d1#ev):= FALSE ;
DEFINE d1#set := (s1#ev) ;
DEFINE d1#reset := (r1#ev) ;
DEFINE d1#ev_next := !d1#val & d1#set | d1#val & d1#reset ;
ASSIGN next(d1#ev) := d1#ev_next ;
ASSIGN next(d1#val) := case
                d1#set: TRUE ;
                d1#reset: FALSE ;
                TRUE: d1#val ;
         esac ;
DEFINE d1#violation1 := d1#set & d1#reset ;
DEFINE d1#set_because#s1 := (s1#ev) ;
DEFINE d1#reset_because#s1 := FALSE ;
DEFINE d1#ev_next_because#s1 := !d1#val & d1#set_because#s1 | d1#val &
   d1#reset_because#s1 ;
DEFINE d1#set_because#r1 := FALSE ;
DEFINE d1#reset_because#r1 := (r1#ev) ;
DEFINE d1#ev_next_because#r1 := !d1#val & d1#set_because#r1 | d1#val &
   d1#reset_because#r1 ;

--- Gate clk_out ---
DEFINE clk_out#violation := clk_out#violation1 ;
VAR clk_out#ev : boolean ;
```

```
ASSIGN init(clk_out#ev) := FALSE ;
DEFINE clk_out#ev_next := (done_0_1#ev) | (s2#ev) | (r2#ev) ;
ASSIGN next(clk_out#ev) := clk_out#ev_next ;
DEFINE clk_out#violation1 := FALSE ;
DEFINE clk_out#ev_next_because#done_0_1 := (done_0_1#ev) ;
DEFINE clk_out#ev_next_because#s2 := (s2#ev) ;
DEFINE clk_out#ev_next_because#r2 := (r2#ev) ;


--- Gate d2 ---
DEFINE d2#violation := d2#violation1 ;
VAR d2#val : boolean ;
VAR d2#ev : boolean ;
ASSIGN init(d2#ev):= FALSE ;
DEFINE d2#set := (s2#ev) ;
DEFINE d2#reset := (r2#ev) ;
DEFINE d2#ev_next := !d2#val & d2#set | d2#val & d2#reset ;
ASSIGN next(d2#ev) := d2#ev_next ;
ASSIGN next(d2#val) := case
                d2#set: TRUE ;
                d2#reset: FALSE ;
                TRUE: d2#val ;
         esac ;
DEFINE d2#violation1 := d2#set & d2#reset ;
DEFINE d2#set_because#s2 := (s2#ev) ;
DEFINE d2#reset_because#s2 := FALSE ;
DEFINE d2#ev_next_because#s2 := !d2#val & d2#set_because#s2 | d2#val &
   d2#reset_because#s2 ;
DEFINE d2#set_because#r2 := FALSE ;
DEFINE d2#reset_because#r2 := (r2#ev) ;
DEFINE d2#ev_next_because#r2 := !d2#val & d2#set_because#r2 | d2#val &
   d2#reset_because#r2 ;
```

```
--- Input Signals ---
VAR clk#ev : boolean ;

--- Input Phrase input_phrase ---
VAR input#state#input_state : boolean ;
ASSIGN init(input#state#input_state) := (TRUE union FALSE) ;
ASSIGN next(input#state#input_state) := case
                input#state#__phrase0_prelude: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state | (clk#ev) ;
VAR input#state#input_state__1 : boolean ;
ASSIGN init(input#state#input_state__1) := FALSE ;
ASSIGN next(input#state#input_state__1) := case
                input#state#input_state: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__1 | (!clk#ev) ;
VAR input#state#input_state__2 : boolean ;
ASSIGN init(input#state#input_state__2) := FALSE ;
ASSIGN next(input#state#input_state__2) := case
                input#state#input_state__1: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__2 | (!clk#ev) ;
VAR input#state#input_state__3 : boolean ;
ASSIGN init(input#state#input_state__3) := FALSE ;
ASSIGN next(input#state#input_state__3) := case
                input#state#input_state__2: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__3 | (!clk#ev) ;
VAR input#state#input_state__4 : boolean ;
ASSIGN init(input#state#input_state__4) := FALSE ;
ASSIGN next(input#state#input_state__4) := case
                input#state#input_state__3: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__4 | (!clk#ev) ;
VAR input#state#input_state__5 : boolean ;
ASSIGN init(input#state#input_state__5) := FALSE ;
ASSIGN next(input#state#input_state__5) := case
                input#state#input_state__4: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__5 | (clk#ev) ;
VAR input#state#input_state__6 : boolean ;
ASSIGN init(input#state#input_state__6) := FALSE ;
ASSIGN next(input#state#input_state__6) := case
                input#state#input_state__5: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__6 | (!clk#ev) ;
VAR input#state#input_state__7 : boolean ;
ASSIGN init(input#state#input_state__7) := FALSE ;
ASSIGN next(input#state#input_state__7) := case
                input#state#input_state__6: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__7 | (!clk#ev) ;
```

```
VAR input#state#input_state__8 : boolean ;
ASSIGN init(input#state#input_state__8) := FALSE ;
ASSIGN next(input#state#input_state__8) := case
                input#state#input_state__7: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__8 | (!clk#ev) ;
VAR input#state#input_state__9 : boolean ;
ASSIGN init(input#state#input_state__9) := FALSE ;
ASSIGN next(input#state#input_state__9) := case
                input#state#input_state__8: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__9 | (!clk#ev) ;
VAR input#state#input_state__10 : boolean ;
ASSIGN init(input#state#input_state__10) := FALSE ;
ASSIGN next(input#state#input_state__10) := case
                input#state#input_state__9: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__10 | (!clk#ev) ;
VAR input#state#__phrase0_prelude : boolean ;
ASSIGN init(input#state#__phrase0_prelude) := FALSE ;
ASSIGN next(input#state#__phrase0_prelude) := case
                input#state#__phrase0_prelude: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#__phrase0_prelude | (!clk#ev) ;
VAR input#state#__phrase0_done : boolean ;
ASSIGN init(input#state#__phrase0_done) := FALSE ;
ASSIGN next(input#state#__phrase0_done) := case
```

```
                input#state#input_state__10: (TRUE union FALSE) ;
                input#state#__phrase0_done: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#__phrase0_done | (!clk#ev) ;
DEFINE meta#input_done := input#state#__phrase0_done ;

INVAR input#state#input_state | input#state#input_state__1 | input#
   state#input_state__2 | input#state#input_state__3 | input#state#
   input_state__4 | input#state#input_state__5 | input#state#
   input_state__6 | input#state#input_state__7 | input#state#
   input_state__8 | input#state#input_state__9 | input#state#
   input_state__10 | input#state#__phrase0_prelude | input#state#
   __phrase0_done ;
INVAR !input#state#input_state | !input#state#input_state__1 ;
INVAR !input#state#input_state | !input#state#input_state__2 ;
INVAR !input#state#input_state | !input#state#input_state__3 ;
INVAR !input#state#input_state | !input#state#input_state__4 ;
INVAR !input#state#input_state | !input#state#input_state__5 ;
INVAR !input#state#input_state | !input#state#input_state__6 ;
INVAR !input#state#input_state | !input#state#input_state__7 ;
INVAR !input#state#input_state | !input#state#input_state__8 ;
INVAR !input#state#input_state | !input#state#input_state__9 ;
INVAR !input#state#input_state | !input#state#input_state__10 ;
INVAR !input#state#input_state | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state | !input#state#__phrase0_done ;
INVAR !input#state#input_state__1 | !input#state#input_state__2 ;
INVAR !input#state#input_state__1 | !input#state#input_state__3 ;
INVAR !input#state#input_state__1 | !input#state#input_state__4 ;
INVAR !input#state#input_state__1 | !input#state#input_state__5 ;
INVAR !input#state#input_state__1 | !input#state#input_state__6 ;
INVAR !input#state#input_state__1 | !input#state#input_state__7 ;
INVAR !input#state#input_state__1 | !input#state#input_state__8 ;
INVAR !input#state#input_state__1 | !input#state#input_state__9 ;
INVAR !input#state#input_state__1 | !input#state#input_state__10 ;
INVAR !input#state#input_state__1 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__1 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__2 | !input#state#input_state__3 ;
INVAR !input#state#input_state__2 | !input#state#input_state__4 ;
INVAR !input#state#input_state__2 | !input#state#input_state__5 ;
INVAR !input#state#input_state__2 | !input#state#input_state__6 ;
INVAR !input#state#input_state__2 | !input#state#input_state__7 ;
INVAR !input#state#input_state__2 | !input#state#input_state__8 ;
INVAR !input#state#input_state__2 | !input#state#input_state__9 ;
INVAR !input#state#input_state__2 | !input#state#input_state__10 ;
INVAR !input#state#input_state__2 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__2 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__3 | !input#state#input_state__4 ;
INVAR !input#state#input_state__3 | !input#state#input_state__5 ;
INVAR !input#state#input_state__3 | !input#state#input_state__6 ;
INVAR !input#state#input_state__3 | !input#state#input_state__7 ;
INVAR !input#state#input_state__3 | !input#state#input_state__8 ;
INVAR !input#state#input_state__3 | !input#state#input_state__9 ;
INVAR !input#state#input_state__3 | !input#state#input_state__10 ;
INVAR !input#state#input_state__3 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__3 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__4 | !input#state#input_state__5 ;
INVAR !input#state#input_state__4 | !input#state#input_state__6 ;
INVAR !input#state#input_state__4 | !input#state#input_state__7 ;
INVAR !input#state#input_state__4 | !input#state#input_state__8 ;
INVAR !input#state#input_state__4 | !input#state#input_state__9 ;
INVAR !input#state#input_state__4 | !input#state#input_state__10 ;
INVAR !input#state#input_state__4 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__4 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__5 | !input#state#input_state__6 ;
INVAR !input#state#input_state__5 | !input#state#input_state__7 ;
INVAR !input#state#input_state__5 | !input#state#input_state__8 ;
INVAR !input#state#input_state__5 | !input#state#input_state__9 ;
INVAR !input#state#input_state__5 | !input#state#input_state__10 ;
INVAR !input#state#input_state__5 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__5 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__6 | !input#state#input_state__7 ;
INVAR !input#state#input_state__6 | !input#state#input_state__8 ;
INVAR !input#state#input_state__6 | !input#state#input_state__9 ;
INVAR !input#state#input_state__6 | !input#state#input_state__10 ;
INVAR !input#state#input_state__6 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__6 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__7 | !input#state#input_state__8 ;
INVAR !input#state#input_state__7 | !input#state#input_state__9 ;
INVAR !input#state#input_state__7 | !input#state#input_state__10 ;
INVAR !input#state#input_state__7 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__7 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__8 | !input#state#input_state__9 ;
INVAR !input#state#input_state__8 | !input#state#input_state__10 ;
INVAR !input#state#input_state__8 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__8 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__9 | !input#state#input_state__10 ;
INVAR !input#state#input_state__9 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__9 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__10 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__10 | !input#state#__phrase0_done ;
INVAR !input#state#__phrase0_prelude | !input#state#__phrase0_done ;
```

```
DEFINE meta#outputs_alive := TRUE ;
DEFINE meta#outputs_accept := TRUE ;

--- Defines to help violation checking ---
DEFINE meta#no_violation := !(s1#violation|r1#violation|done_0_1#
   violation|r2#violation|s0#violation|clk_out#violation|r0#violation|
   s2#violation|done_0#violation|d1#violation|d0#violation|d2#violation
   ) ;
```

B.4 “Speed” Model

```
MODULE main

--- Gate s0 ---
DEFINE s0#violation := (clk#ev&d0#ev)|s0#violation1 ;
VAR s0#ev : boolean ;
DEFINE s0#ev_next := (clk#ev & !d0#val) ;
ASSIGN next(s0#ev) := s0#ev_next ;
DEFINE s0#violation1 := FALSE ;
DEFINE s0#ev_next_because#clk := (clk#ev & !d0#val) ;
DEFINE s0#ev_next_fast := (clk#ev & clk#ev_fast & !d0#val) ;
VAR s0#ev_fast : boolean ;
ASSIGN next(s0#ev_fast) := (meta#apply_speed_cur & s0#ev_next) ? s0#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate r0 ---
DEFINE r0#violation := (clk#ev&d0#ev)|r0#violation1 ;
VAR r0#ev : boolean ;
DEFINE r0#ev_next := (clk#ev & d0#val) ;
ASSIGN next(r0#ev) := r0#ev_next ;
DEFINE r0#violation1 := FALSE ;
DEFINE r0#ev_next_because#clk := (clk#ev & d0#val) ;
DEFINE r0#ev_next_fast := (clk#ev & clk#ev_fast & d0#val) ;
VAR r0#ev_fast : boolean ;
ASSIGN next(r0#ev_fast) := (meta#apply_speed_cur & r0#ev_next) ? r0#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate s1 ---
DEFINE s1#violation := (r0#ev&d1#ev)|s1#violation1 ;
VAR s1#ev : boolean ;
DEFINE s1#ev_next := (r0#ev & !d1#val) ;
ASSIGN next(s1#ev) := s1#ev_next ;
DEFINE s1#violation1 := FALSE ;
DEFINE s1#ev_next_because#r0 := (r0#ev & !d1#val) ;
DEFINE s1#ev_next_fast := (r0#ev & r0#ev_fast & !d1#val) ;
VAR s1#ev_fast : boolean ;
ASSIGN next(s1#ev_fast) := (meta#apply_speed_cur & s1#ev_next) ? s1#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate r1 ---
DEFINE r1#violation := (r0#ev&d1#ev)|r1#violation1 ;
VAR r1#ev : boolean ;
DEFINE r1#ev_next := (r0#ev & d1#val) ;
ASSIGN next(r1#ev) := r1#ev_next ;
DEFINE r1#violation1 := FALSE ;
DEFINE r1#ev_next_because#r0 := (r0#ev & d1#val) ;
DEFINE r1#ev_next_fast := (r0#ev & r0#ev_fast & d1#val) ;
VAR r1#ev_fast : boolean ;
ASSIGN next(r1#ev_fast) := (meta#apply_speed_cur & r1#ev_next) ? r1#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate done_0 ---
DEFINE done_0#violation := done_0#violation1 ;
VAR done_0#ev : boolean ;
DEFINE done_0#ev_next := (s0#ev) ;
```

```
ASSIGN next(done_0#ev) := done_0#ev_next ;
DEFINE done_0#violation1 := FALSE ;
DEFINE done_0#ev_next_because#s0 := (s0#ev) ;
DEFINE done_0#ev_next_fast := (s0#ev & s0#ev_fast) ;
VAR done_0#ev_fast : boolean ;
ASSIGN next(done_0#ev_fast) := (meta#apply_speed_cur & done_0#ev_next)
   ? done_0#ev_next_fast: (TRUE union FALSE) ;
--- Gate d0 ---
DEFINE d0#violation := d0#violation1 ;
VAR d0#val : boolean ;
VAR d0#ev : boolean ;
DEFINE d0#set := (s0#ev) ;
DEFINE d0#reset := (r0#ev) ;
DEFINE d0#ev_next := !d0#val & d0#set | d0#val & d0#reset ;
ASSIGN next(d0#ev) := d0#ev_next ;
ASSIGN next(d0#val) := case
                d0#set: TRUE ;
                d0#reset: FALSE ;
                TRUE: d0#val ;
         esac ;
DEFINE d0#violation1 := d0#set & d0#reset ;
DEFINE d0#set_because#s0 := (s0#ev) ;
DEFINE d0#reset_because#s0 := FALSE ;
DEFINE d0#ev_next_because#s0 := !d0#val & d0#set_because#s0 | d0#val &
   d0#reset_because#s0 ;
DEFINE d0#set_because#r0 := FALSE ;
DEFINE d0#reset_because#r0 := (r0#ev) ;
DEFINE d0#ev_next_because#r0 := !d0#val & d0#set_because#r0 | d0#val &
   d0#reset_because#r0 ;
DEFINE d0#set_fast := (s0#ev & s0#ev_fast) ;
DEFINE d0#reset_fast := (r0#ev & r0#ev_fast) ;
DEFINE d0#ev_next_fast := !d0#val & d0#set_fast | d0#val & d0#
   reset_fast ;
VAR d0#ev_fast : boolean ;
ASSIGN next(d0#ev_fast) := (meta#apply_speed_cur & d0#ev_next) ? d0#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate s2 ---
DEFINE s2#violation := (r1#ev&d2#ev)|s2#violation1 ;
VAR s2#ev : boolean ;
DEFINE s2#ev_next := (r1#ev & !d2#val) ;
ASSIGN next(s2#ev) := s2#ev_next ;
DEFINE s2#violation1 := FALSE ;
DEFINE s2#ev_next_because#r1 := (r1#ev & !d2#val) ;
DEFINE s2#ev_next_fast := (r1#ev & r1#ev_fast & !d2#val) ;
VAR s2#ev_fast : boolean ;
ASSIGN next(s2#ev_fast) := (meta#apply_speed_cur & s2#ev_next) ? s2#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate r2 ---
DEFINE r2#violation := (r1#ev&d2#ev)|r2#violation1 ;
VAR r2#ev : boolean ;
DEFINE r2#ev_next := (r1#ev & d2#val) ;
ASSIGN next(r2#ev) := r2#ev_next ;
DEFINE r2#violation1 := FALSE ;
DEFINE r2#ev_next_because#r1 := (r1#ev & d2#val) ;
DEFINE r2#ev_next_fast := (r1#ev & r1#ev_fast & d2#val) ;
VAR r2#ev_fast : boolean ;
ASSIGN next(r2#ev_fast) := (meta#apply_speed_cur & r2#ev_next) ? r2#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate done_0_1 ---
DEFINE done_0_1#violation := done_0_1#violation1 ;
VAR done_0_1#ev : boolean ;
DEFINE done_0_1#ev_next := (done_0#ev) | (s1#ev) ;
```

```
ASSIGN next(done_0_1#ev) := done_0_1#ev_next ;
DEFINE done_0_1#violation1 := FALSE ;
DEFINE done_0_1#ev_next_because#done_0 := (done_0#ev) ;
DEFINE done_0_1#ev_next_because#s1 := (s1#ev) ;
DEFINE done_0_1#ev_next_fast := (done_0#ev & done_0#ev_fast) | (s1#ev &
    s1#ev_fast) ;
VAR done_0_1#ev_fast : boolean ;
ASSIGN next(done_0_1#ev_fast) := (meta#apply_speed_cur & done_0_1#
   ev_next) ? done_0_1#ev_next_fast: (TRUE union FALSE) ;
--- Gate d1 ---
DEFINE d1#violation := d1#violation1 ;
VAR d1#val : boolean ;
VAR d1#ev : boolean ;
DEFINE d1#set := (s1#ev) ;
DEFINE d1#reset := (r1#ev) ;
DEFINE d1#ev_next := !d1#val & d1#set | d1#val & d1#reset ;
ASSIGN next(d1#ev) := d1#ev_next ;
ASSIGN next(d1#val) := case
                d1#set: TRUE ;
                d1#reset: FALSE ;
                TRUE: d1#val ;
         esac ;
DEFINE d1#violation1 := d1#set & d1#reset ;
DEFINE d1#set_because#s1 := (s1#ev) ;
DEFINE d1#reset_because#s1 := FALSE ;
DEFINE d1#ev_next_because#s1 := !d1#val & d1#set_because#s1 | d1#val &
   d1#reset_because#s1 ;
DEFINE d1#set_because#r1 := FALSE ;
DEFINE d1#reset_because#r1 := (r1#ev) ;
DEFINE d1#ev_next_because#r1 := !d1#val & d1#set_because#r1 | d1#val &
   d1#reset_because#r1 ;
```

```
DEFINE d1#set_fast := (s1#ev & s1#ev_fast) ;
DEFINE d1#reset_fast := (r1#ev & r1#ev_fast) ;
DEFINE d1#ev_next_fast := !d1#val & d1#set_fast | d1#val & d1#
   reset_fast ;
VAR d1#ev_fast : boolean ;
ASSIGN next(d1#ev_fast) := (meta#apply_speed_cur & d1#ev_next) ? d1#
   ev_next_fast: (TRUE union FALSE) ;
--- Gate clk_out ---
DEFINE clk_out#violation := clk_out#violation1 ;
VAR clk_out#ev : boolean ;
DEFINE clk_out#ev_next := (done_0_1#ev) | (s2#ev) | (r2#ev) ;
ASSIGN next(clk_out#ev) := clk_out#ev_next ;
DEFINE clk_out#violation1 := FALSE ;
DEFINE clk_out#ev_next_because#done_0_1 := (done_0_1#ev) ;
DEFINE clk_out#ev_next_because#s2 := (s2#ev) ;
DEFINE clk_out#ev_next_because#r2 := (r2#ev) ;
DEFINE clk_out#ev_next_fast := (done_0_1#ev & done_0_1#ev_fast) | (s2#
   ev & s2#ev_fast) | (r2#ev & r2#ev_fast) ;
VAR clk_out#ev_fast : boolean ;
ASSIGN next(clk_out#ev_fast) := (meta#apply_speed_cur & clk_out#ev_next
   ) ? clk_out#ev_next_fast: (TRUE union FALSE) ;
--- Gate d2 ---
DEFINE d2#violation := d2#violation1 ;
VAR d2#val : boolean ;
VAR d2#ev : boolean ;
DEFINE d2#set := (s2#ev) ;
DEFINE d2#reset := (r2#ev) ;
DEFINE d2#ev_next := !d2#val & d2#set | d2#val & d2#reset ;
ASSIGN next(d2#ev) := d2#ev_next ;
ASSIGN next(d2#val) := case
                d2#set: TRUE ;
                d2#reset: FALSE ;
                TRUE: d2#val ;
         esac ;
DEFINE d2#violation1 := d2#set & d2#reset ;
DEFINE d2#set_because#s2 := (s2#ev) ;
DEFINE d2#reset_because#s2 := FALSE ;
DEFINE d2#ev_next_because#s2 := !d2#val & d2#set_because#s2 | d2#val &
   d2#reset_because#s2 ;
DEFINE d2#set_because#r2 := FALSE ;
DEFINE d2#reset_because#r2 := (r2#ev) ;
DEFINE d2#ev_next_because#r2 := !d2#val & d2#set_because#r2 | d2#val &
   d2#reset_because#r2 ;
DEFINE d2#set_fast := (s2#ev & s2#ev_fast) ;
DEFINE d2#reset_fast := (r2#ev & r2#ev_fast) ;
DEFINE d2#ev_next_fast := !d2#val & d2#set_fast | d2#val & d2#
   reset_fast ;
VAR d2#ev_fast : boolean ;
ASSIGN next(d2#ev_fast) := (meta#apply_speed_cur & d2#ev_next) ? d2#
   ev_next_fast: (TRUE union FALSE) ;

--- Input Signals ---
VAR clk#ev : boolean ;
VAR clk#ev_fast : boolean ;
INVAR !meta#input_coherent | !clk#ev | (clk#ev_fast = meta#input_fast)
   ;

--- Input coherence ---
VAR meta#input_fast: boolean ;
ASSIGN next(meta#input_fast) := meta#input_fast ;
VAR meta#input_coherence_count_1 : boolean ;
ASSIGN init(meta#input_coherence_count_1) := FALSE ;
ASSIGN next(meta#input_coherence_count_1) := meta#
    input_coherence_count_1 | meta#apply_speed_cur ;
DEFINE meta#input_coherent := meta#input_coherence_count_1 ;
--- Input Phrase input_phrase ---
VAR input#state#input_state : boolean ;
ASSIGN next(input#state#input_state) := case
                input#state#__phrase0_prelude: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state | (clk#ev) ;
VAR input#state#input_state__1 : boolean ;
ASSIGN next(input#state#input_state__1) := case
                input#state#input_state: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__1 | (!clk#ev) ;
VAR input#state#input_state__2 : boolean ;
ASSIGN next(input#state#input_state__2) := case
                input#state#input_state__1: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__2 | (!clk#ev) ;
VAR input#state#input_state__3 : boolean ;
ASSIGN next(input#state#input_state__3) := case
                input#state#input_state__2: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__3 | (!clk#ev) ;
VAR input#state#input_state__4 : boolean ;
ASSIGN next(input#state#input_state__4) := case
                input#state#input_state__3: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__4 | (!clk#ev) ;
VAR input#state#input_state__5 : boolean ;
ASSIGN next(input#state#input_state__5) := case
                input#state#input_state__4: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__5 | (clk#ev) ;
VAR input#state#input_state__6 : boolean ;
ASSIGN next(input#state#input_state__6) := case
                input#state#input_state__5: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__6 | (!clk#ev) ;
VAR input#state#input_state__7 : boolean ;
ASSIGN next(input#state#input_state__7) := case
                input#state#input_state__6: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__7 | (!clk#ev) ;
VAR input#state#input_state__8 : boolean ;
ASSIGN next(input#state#input_state__8) := case
                input#state#input_state__7: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__8 | (!clk#ev) ;
VAR input#state#input_state__9 : boolean ;
ASSIGN next(input#state#input_state__9) := case
                input#state#input_state__8: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__9 | (!clk#ev) ;
VAR input#state#input_state__10 : boolean ;
ASSIGN next(input#state#input_state__10) := case
                input#state#input_state__9: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#input_state__10 | (!clk#ev) ;
VAR input#state#__phrase0_prelude : boolean ;
ASSIGN next(input#state#__phrase0_prelude) := case
                input#state#__phrase0_prelude: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#__phrase0_prelude | (!clk#ev) ;
VAR input#state#__phrase0_done : boolean ;
ASSIGN next(input#state#__phrase0_done) := case
                input#state#input_state__10: (TRUE union FALSE) ;
                input#state#__phrase0_done: (TRUE union FALSE) ;
                TRUE: FALSE ;
        esac ;
INVAR !input#state#__phrase0_done | (!clk#ev) ;
DEFINE meta#input_done := input#state#__phrase0_done ;

INVAR input#state#input_state | input#state#input_state__1 | input#
   state#input_state__2 | input#state#input_state__3 | input#state#
   input_state__4 | input#state#input_state__5 | input#state#
   input_state__6 | input#state#input_state__7 | input#state#
   input_state__8 | input#state#input_state__9 | input#state#
   input_state__10 | input#state#__phrase0_prelude | input#state#
   __phrase0_done ;
INVAR !input#state#input_state | !input#state#input_state__1 ;
INVAR !input#state#input_state | !input#state#input_state__2 ;
INVAR !input#state#input_state | !input#state#input_state__3 ;
INVAR !input#state#input_state | !input#state#input_state__4 ;
INVAR !input#state#input_state | !input#state#input_state__5 ;
INVAR !input#state#input_state | !input#state#input_state__6 ;
INVAR !input#state#input_state | !input#state#input_state__7 ;
INVAR !input#state#input_state | !input#state#input_state__8 ;
INVAR !input#state#input_state | !input#state#input_state__9 ;
INVAR !input#state#input_state | !input#state#input_state__10 ;
INVAR !input#state#input_state | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state | !input#state#__phrase0_done ;
INVAR !input#state#input_state__1 | !input#state#input_state__2 ;
INVAR !input#state#input_state__1 | !input#state#input_state__3 ;
INVAR !input#state#input_state__1 | !input#state#input_state__4 ;
INVAR !input#state#input_state__1 | !input#state#input_state__5 ;
INVAR !input#state#input_state__1 | !input#state#input_state__6 ;
INVAR !input#state#input_state__1 | !input#state#input_state__7 ;
INVAR !input#state#input_state__1 | !input#state#input_state__8 ;
INVAR !input#state#input_state__1 | !input#state#input_state__9 ;
INVAR !input#state#input_state__1 | !input#state#input_state__10 ;
INVAR !input#state#input_state__1 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__1 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__2 | !input#state#input_state__3 ;
INVAR !input#state#input_state__2 | !input#state#input_state__4 ;
INVAR !input#state#input_state__2 | !input#state#input_state__5 ;
INVAR !input#state#input_state__2 | !input#state#input_state__6 ;
INVAR !input#state#input_state__2 | !input#state#input_state__7 ;
INVAR !input#state#input_state__2 | !input#state#input_state__8 ;
INVAR !input#state#input_state__2 | !input#state#input_state__9 ;
INVAR !input#state#input_state__2 | !input#state#input_state__10 ;
INVAR !input#state#input_state__2 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__2 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__3 | !input#state#input_state__4 ;
INVAR !input#state#input_state__3 | !input#state#input_state__5 ;
INVAR !input#state#input_state__3 | !input#state#input_state__6 ;
INVAR !input#state#input_state__3 | !input#state#input_state__7 ;
INVAR !input#state#input_state__3 | !input#state#input_state__8 ;
INVAR !input#state#input_state__3 | !input#state#input_state__9 ;
INVAR !input#state#input_state__3 | !input#state#input_state__10 ;
INVAR !input#state#input_state__3 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__3 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__4 | !input#state#input_state__5 ;
INVAR !input#state#input_state__4 | !input#state#input_state__6 ;
INVAR !input#state#input_state__4 | !input#state#input_state__7 ;
INVAR !input#state#input_state__4 | !input#state#input_state__8 ;
INVAR !input#state#input_state__4 | !input#state#input_state__9 ;
INVAR !input#state#input_state__4 | !input#state#input_state__10 ;
INVAR !input#state#input_state__4 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__4 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__5 | !input#state#input_state__6 ;
INVAR !input#state#input_state__5 | !input#state#input_state__7 ;
INVAR !input#state#input_state__5 | !input#state#input_state__8 ;
INVAR !input#state#input_state__5 | !input#state#input_state__9 ;
INVAR !input#state#input_state__5 | !input#state#input_state__10 ;
INVAR !input#state#input_state__5 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__5 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__6 | !input#state#input_state__7 ;
INVAR !input#state#input_state__6 | !input#state#input_state__8 ;
INVAR !input#state#input_state__6 | !input#state#input_state__9 ;
INVAR !input#state#input_state__6 | !input#state#input_state__10 ;
INVAR !input#state#input_state__6 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__6 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__7 | !input#state#input_state__8 ;
INVAR !input#state#input_state__7 | !input#state#input_state__9 ;
INVAR !input#state#input_state__7 | !input#state#input_state__10 ;
INVAR !input#state#input_state__7 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__7 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__8 | !input#state#input_state__9 ;
INVAR !input#state#input_state__8 | !input#state#input_state__10 ;
INVAR !input#state#input_state__8 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__8 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__9 | !input#state#input_state__10 ;
INVAR !input#state#input_state__9 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__9 | !input#state#__phrase0_done ;
INVAR !input#state#input_state__10 | !input#state#__phrase0_prelude ;
INVAR !input#state#input_state__10 | !input#state#__phrase0_done ;
INVAR !input#state#__phrase0_prelude | !input#state#__phrase0_done ;


DEFINE meta#outputs_alive := TRUE ;
DEFINE meta#outputs_accept := TRUE ;
--- Start from all reachable states ---
INIT (!input#state#__phrase0_prelude & case
        s0#ev : ((((((((((((((((((((((!r0#ev & !s1#ev) & __expr0) &
            __expr1) & __expr2) & __expr3) & __expr4) & __expr5) &
            __expr6) & __expr7) & __expr8) & __expr9) & __expr10) &
            __expr11) & __expr12) & __expr13) & __expr14) & __expr15) &
            __expr16) & __expr17) & __expr18) & __expr19) & __expr20) &
            __expr21);
        r0#ev : ((((((((((((((((((((!s1#ev & __expr0) & __expr1) & d0
            #val) & __expr3) & __expr4) & __expr5) & __expr6) & __expr7)
             & __expr8) & __expr9) & __expr10) & __expr11) & __expr12) &
             __expr13) & __expr14) & __expr15) & __expr16) & __expr17) &
```

```
     __expr18) & __expr19) & __expr20) & __expr21);
TRUE : ((!input#state#input_state__1 & !input#state#
    input_state__6) & case
s1#ev : ((((((((((((((((((__expr0 & __expr1) & __expr2) & d0#
    ev) & __expr4) & __expr5) & __expr6) & __expr22) & __expr7)
    & __expr8) & __expr9) & __expr10) & __expr11) & __expr13) &
    __expr14) & __expr15) & __expr17) & __expr18) & __expr19) &
    __expr20) & __expr23);
r1#ev : (((((((((((((((((__expr1 & __expr2) & d0#ev) &
    __expr4) & __expr5) & __expr6) & d1#val) & __expr7) &
    __expr8) & __expr9) & __expr10) & __expr11) & __expr13) &
    __expr14) & __expr15) & __expr17) & __expr18) & __expr19) &
    __expr20) & __expr23);
done_0#ev : ((((((((((((((((d0#val & d0#ev) & __expr4) &
    __expr5) & __expr6) & __expr7) & __expr8) & __expr9) &
    __expr10) & __expr11) & __expr13) & __expr14) & __expr15) &
    __expr17) & __expr18) & __expr19) & __expr20) & __expr23);
TRUE : (((__expr3 & __expr12) & __expr16) & case
d0#val : ((((__expr4 & __expr5) & __expr7) & __expr9) & case
done_0_1#ev : ((((((((__expr8 & __expr10) & __expr11) &
    __expr14) & __expr15) & __expr18) & __expr19) & __expr20) &
    __expr24);
TRUE : ((__expr13 & __expr17) & __expr27);
esac);
s2#ev : (((((((((((((__expr5 & __expr6) & __expr22) & d1#ev) &
     __expr8) & !d2#val) & __expr9) & __expr10) & __expr11) &
    __expr14) & __expr15) & __expr18) & __expr19) & __expr20) &
    __expr24);
r2#ev : ((((((((((((__expr6 & __expr22) & d1#ev) & __expr8) &
    d2#val) & __expr9) & __expr10) & __expr11) & __expr14) &
    __expr15) & __expr18) & __expr19) & __expr20) & __expr24);
```

```
        done_0_1#ev : (((((((((((d1#val & d1#ev) & __expr8) & __expr9)
           & __expr10) & __expr11) & __expr14) & __expr15) & __expr18)
           & __expr19) & __expr20) & __expr24);
        TRUE : (((__expr7 & __expr13) & __expr17) & case
        d1#val : (__expr9 & __expr27);
        clk_out#ev : ((((((d2#ev & __expr10) & __expr11) & __expr15) &
           __expr19) & __expr20) & __expr25);
        TRUE : (((__expr9 & __expr14) & __expr18) & __expr26);
        esac);
        esac);
        esac);
        esac) ;
DEFINE __expr10 := !clk#ev ;
DEFINE __expr27 := case
clk_out#ev : (((((__expr10 & __expr11) & __expr15) & __expr19) &
   __expr20) & __expr25);
TRUE : ((__expr14 & __expr18) & __expr26);
esac ;
DEFINE __expr11 := !input#state#input_state ;
DEFINE __expr24 := case
input#state#input_state__3 : __expr17;
TRUE : input#state#input_state__8;
esac ;
DEFINE __expr1 := !done_0#ev ;
DEFINE __expr3 := !d0#ev ;
DEFINE __expr16 := !input#state#input_state__7 ;
DEFINE __expr0 := !r1#ev ;
DEFINE __expr15 := !input#state#input_state__5 ;
DEFINE __expr6 := !done_0_1#ev ;
DEFINE __expr5 := !r2#ev ;
DEFINE __expr8 := !clk_out#ev ;
DEFINE __expr18 := !input#state#input_state__9 ;
DEFINE __expr2 := !d0#val ;
DEFINE __expr19 := !input#state#input_state__10 ;
DEFINE __expr20 := !input#state#__phrase0_done ;
DEFINE __expr23 := case
input#state#input_state__2 : __expr16;
TRUE : input#state#input_state__7;
esac ;
DEFINE __expr22 := !d1#val ;
DEFINE __expr17 := !input#state#input_state__8 ;
DEFINE __expr13 := !input#state#input_state__3 ;
DEFINE __expr9 := !d2#ev ;
DEFINE __expr25 := case
input#state#input_state__4 : __expr18;
TRUE : input#state#input_state__9;
esac ;
DEFINE __expr26 := case
clk#ev : ((__expr19 & __expr20) & case
input#state#input_state : __expr15;
TRUE : input#state#input_state__5;
esac);
TRUE : ((__expr11 & __expr15) & case
input#state#input_state__10 : __expr20;
TRUE : input#state#__phrase0_done;
esac);
esac ;
DEFINE __expr12 := !input#state#input_state__2 ;
DEFINE __expr14 := !input#state#input_state__4 ;
DEFINE __expr21 := case
input#state#input_state__1 : !input#state#input_state__6;
TRUE : input#state#input_state__6;
```

```
esac ;
DEFINE __expr7 := !d1#ev ;
DEFINE __expr4 := !s2#ev ;

--- Defines to help cohrence depth testing ---
DEFINE meta#signals_coherent_fast := (!s1#ev| s1#ev_fast) & (!r1#ev| r1
   #ev_fast) & (!done_0_1#ev| done_0_1#ev_fast) & (!r2#ev| r2#ev_fast)
   & (!s0#ev| s0#ev_fast) & (!clk_out#ev| clk_out#ev_fast) & (!r0#ev|
   r0#ev_fast) & (!s2#ev| s2#ev_fast) & (!clk#ev| clk#ev_fast) & (!
   done_0#ev| done_0#ev_fast) & (!d1#ev| d1#ev_fast) & (!d0#ev| d0#
   ev_fast) & (!d2#ev| d2#ev_fast) ;
DEFINE meta#signals_coherent_slow := (!s1#ev|!s1#ev_fast) & (!r1#ev|!r1
   #ev_fast) & (!done_0_1#ev|!done_0_1#ev_fast) & (!r2#ev|!r2#ev_fast)
   & (!s0#ev|!s0#ev_fast) & (!clk_out#ev|!clk_out#ev_fast) & (!r0#ev|!
   r0#ev_fast) & (!s2#ev|!s2#ev_fast) & (!clk#ev|!clk#ev_fast) & (!
   done_0#ev|!done_0#ev_fast) & (!d1#ev|!d1#ev_fast) & (!d0#ev|!d0#
   ev_fast) & (!d2#ev|!d2#ev_fast) ;
DEFINE meta#input_coherent_fast := meta#input_done | (meta#
   input_coherent & meta#input_fast) ;
DEFINE meta#input_coherent_slow := meta#input_done | (meta#
   input_coherent & !meta#input_fast) ;
DEFINE meta#coherent_fast := meta#signals_coherent_fast & meta#
   input_coherent_fast ;
DEFINE meta#coherent_slow := meta#signals_coherent_slow & meta#
   input_coherent_slow ;
DEFINE meta#is_coherent := meta#coherent_fast | meta#coherent_slow ;

--- Variables to govern application of speed labelling ---
VAR meta#apply_speed_cur: boolean ;
VAR meta#apply_speed_next: boolean ;
INVAR !meta#apply_speed_cur | meta#apply_speed_next ;
```

```
ASSIGN next(meta#apply_speed_cur) := meta#apply_speed_next ;

--- Defines to help violation checking ---
DEFINE meta#no_violation := !(s1#violation|r1#violation|done_0_1#
    violation|r2#violation|s0#violation|clk_out#violation|r0#violation|
    s2#violation|done_0#violation|d1#violation|d0#violation|d2#violation
    ) ;
```

Bibliography

[1] M. Miller, *Realization and formal analysis of asynchronous pulse communication circuits.* PhD thesis, Dept. Elec. and Computer Engineering, Univ. California Santa Barbara, 2015.

[2] M. Miller, C. Segal, D. Mc Carthy, A. Dalakoti, P. Mukim, and F. Brewer, *Impolite high speed interfaces with asynchronous pulse logic*, in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18, (New York, NY, USA), pp. 99–104, ACM, 2018.

[3]

[4] M. Miller, G. Hoover, and F. Brewer, *Pulse-mode link for robust, high speed communications*, in *2008 IEEE International Symposium on Circuits and Systems*, pp. 3073–3077, May, 2008.

[5] A. Dalakoti, *Collective Pulse Dynamics: A New Timing Circuit Strategy.* University of California, Santa Barbara, 2018.

[6] P. Mukim, *Analysis and Design of Precision Timing Circuits using Pulse Mode Event Signaling.* PhD thesis, UC Santa Barbara, 2020.

[7] P. Mukim, A. Dalakoti, D. McCarthy, C. Segal, M. Miller, J. F. Buckwalter, and F. Brewer, *Design and analysis of collective pulse oscillators*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **28** (2019), no. 5 1242–1255.

[8] P. Mukim and F. Brewer, *Multiwire phase encoding: A signaling strategy for high-bandwidth, low-power data movement*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2021).

[9] F. Brewer. Private Communication.

[10] M. Nyström, *Asynchronous pulse logic.* Kluwer Academic Publishers, Boston, 2002.

[11] I. Sutherland and S. Fairbanks, *Gasp: A minimal fifo control*, in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pp. 46–53, IEEE, 2001.

[12] M. R. Greenstreet, *Surfing interconnect*, in *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)*, pp. 9 pp.–106, March, 2006.

[13] V. Narayanan, B. A. Chappell, and B. M. Fleischer, *Static timing analysis for self resetting circuits*, in *Proceedings of International Conference on Computer Aided Design*, pp. 119–126, Nov, 1996.

[14] G. Hinton, M. Upton, D. J. Sager, D. Boggs, D. M. Carmean, P. Roussel, T. I. Chappell, T. D. Fletcher, M. S. Milshtein, M. Sprague, S. Samaan, and R. Murray, *A 0.18-/spl mu/m cmos ia-32 processor with a 4-ghz integer execution unit*, *IEEE Journal of Solid-State Circuits* **36** (Nov, 2001) 1617–1627.

[15] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms.* Springer Science & Business Media, 2007.

[16] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, *Symbolic model checking: 1020 states and beyond*, *Information and Computation* **98** (1992), no. 2 142–170.

[17] R. E. Bryant, *Graph-based algorithms for boolean function manipulation*, *IEEE Transactions on Computers* **C-35** (1986), no. 8 677–691.

[18] R. E. Bryant, *On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication*, *IEEE Transactions on Computers* **40** (1991), no. 2 205–213.

[19] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, *Bounded model checking*, .

[20] M. Sheeran, S. Singh, and G. Stålmarck, *Checking safety properties using induction and a sat-solver*, in *International conference on formal methods in computer-aided design*, pp. 127–144, Springer, 2000.

[21] N. Een, A. Mishchenko, and R. Brayton, *Efficient implementation of property directed reachability*, in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 125–134, 2011.

[22] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in *Workshop on Logic of Programs*, pp. 52–71, Springer, 1981.

[23] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking.* MIT press, 2018.

[24] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, *Nusmv: a new symbolic model checker*, *International Journal on Software Tools for Technology Transfer* **2** (2000), no. 4 410–425.

[25] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, *The nuxmv symbolic model checker*, in *International Conference on Computer Aided Verification*, pp. 334–342, Springer, 2014.

[26] D. Kroening and M. Purandare, "Ebmc: The enhanced bounded model checker." online, 2017.

[27] F. Somenzi, *Cudd: Cu decision diagram package-release 2.4. 0*, *University of Colorado at Boulder* (2004).

[28] N. Eén and N. Sörensson, *An extensible sat-solver*, in *International conference on theory and applications of satisfiability testing*, pp. 502–518, Springer, 2003.

[29] A. Biere, *Picosat essentials*, *Journal on Satisfiability, Boolean Modeling and Computation* **4** (2008) 75–97. 2-4.

[30] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, *CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020*, in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions* (T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2020-1 of *Department of Computer Science Report Series B*, pp. 51–53, University of Helsinki, 2020.

[31] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, *Learning rate based branching heuristic for sat solvers*, in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 123–140, Springer, 2016.

[32] D. Huffman, *The synthesis of sequential switching circuits*, *Journal of the Franklin Institute* **257** (1954), no. 3 161–190.

[33] W. S. Coates, A. L. Davis, and K. S. Stevens, *Automatic synthesis of fast compact self-timed control circuits*, in *IFIP Working Conference on Design Methodologies*, pp. 193–208, Citeseer, 1993.

[34] R. M. Fuhrer and S. M. Nowick, *Sequential optimization of asynchronous and synchronous finite-state machines: Algorithms and tools.* Springer Science & Business Media, 2012.

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

[36] S. P. Wilcox, *Synthesis of asynchronous circuits*, tech. rep., University of Cambridge, Computer Laboratory, 1999.

[37] 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** (1997), no. 3 315–325.

[38] K. S. Stevens, R. Ginosar, and S. Rotem, *Relative timing [asynchronous design]*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **11** (Feb, 2003) 129–140.

[39] D. E. Muller, *A theory of asynchronous circuits*, *Report 75, University of Illinois* (1956).

[40] W. A. Clark, *Macromodular computer systems*, in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 335–336, 1967.

[41] P. Patra and D. S. Fussell, *Efficient building blocks for delay insensitive circuits*, in *Proceedings of 1994 IEEE Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 196–205, IEEE, 1994.

[42] A. J. Martin, *The limitations to delay-insensitivity in asynchronous circuits*, in *Beauty is our business*, pp. 302–311. Springer, 1990.

[43] A. J. Martin, M. Nystrom, and C. G. Wong, *Three generations of asynchronous microprocessors*, *IEEE Design & Test of Computers* **20** (2003), no. 6 9–17.

[44] A. J. Martin and M. Nystrom, *Asynchronous techniques for system-on-chip design*, *Proceedings of the IEEE* **94** (2006), no. 6 1089–1120.

[45] C. A. R. Hoare, *Communicating sequential processes*, *Communications of the ACM* **21** (1978), no. 8 666–677.

[46] P. A. Beerel, G. D. Dimou, and A. M. Lines, *Proteus: An asic flow for ghz asynchronous designs*, *IEEE Design Test of Computers* **28** (2011), no. 5 36–51.

[47] I. E. Sutherland, *Micropipelines*, *Communications of the ACM* **32** (1989), no. 6 720–738.

[48] W. Belluomini and C. J. Myers, *Verification of timed systems using posets*, in *International Conference on Computer Aided Verification*, pp. 403–415, Springer, 1998.

[49] R. Alur and D. L. Dill, *A theory of timed automata*, *Theoretical computer science* **126** (1994), no. 2 183–235.

[50] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, *Uppaal—a tool suite for automatic verification of real-time systems*, in *International hybrid systems workshop*, pp. 232–243, Springer, 1995.

[51] O. Maler and A. Pnueli, *Timing analysis of asynchronous circuits using timed automata*, in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp. 189–205, Springer, 1995.

[52] M. D. Riedel, *Cyclic combinational circuits*. California Institute of Technology, 2004.

[53] Python Core Team, "Python: A dynamic, open source programming language." Online, 2019. Python version 3.9.5.

[54] S. Haynal, G. Hoover, A. Vijayakumar, K. Arya, M. Miller, and F. Brewer, "Pycudd." Online, 2015.

[55] D. Beazley, *Writing parsers and compilers with ply*, *PyCon'07* (2007).