ALGORITHMS FOR AUTOMATIC GENERATION OF RELATIVE TIMING CONSTRAINTS

by

Yang Xu

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

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

May 2011

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

This dissertation of **Yang Xu**

has been approved by the following supervisory committee members:

Kenneth S. Stevens, Chair		**03/03/2011** Date Approved
Chris J. Myers, Member		**03/03/2011** Date Approved
Ganesh Gopalakrishnan, Member		**03/01/2011** Date Approved
Priyank Kalla, Member		**03/03/2011** Date Approved
Marly Roncken, Member		**02/23/2011** Date Approved

and by **Gianluca Lazzi**, Chair of the Department of **Electrical and Computer Engineering**

and by Charles A. Wight, Dean of the Graduate School.

ABSTRACT

Asynchronous circuits exhibit impressive power and performance benefits over its synchronous counterpart. Asynchronous system design, however, is not widely adopted due to the fact that it lacks an equivalent support of CAD tools and requires deep expertise in asynchronous circuit design. A relative timing (RT) based asynchronous circuit design flow using traditional synchronous commercial CAD tools was recently proposed. This design flow enables engineers who are proficient in using synchronous design and CAD flow to more easily switch to asynchronous design without asynchronous experience while retaining the asynchronous benefits of power and performance. Relative timing constraints are the key step to this design flow, and were generated manually by the designer based on his/her intuition and understanding of the circuit logic and structure. This process was quite time-consuming and error-prone.

This dissertation presents an algorithm that automatically generates a set of relative timing constraints to guarantee the correctness of a circuit with the aid of a formal verification engine – Analyze. The algorithms have been implemented in a tool called ARTIST (Automatic Relative Timing Identifier based on Signal Traces).

Automatic generation of relative timing constraints relies on manipulation, such as searching and backtracking, of a trace status tableau that is built based on the counter example signal trace returned from the formal verification engine. The underlying mechanism of relative timing is to force signal ordering on the labeled transition graph of the system to restrict its reachability to failure states such that the circuit implementation conforms to the specification. Examples from a simple C-Element to complex six-four GasP circuits are demonstrated to show how this technique is applied to real problems.

The set of relative timing constraints generated by ARTIST is compared against the set of hand generated constraints in terms of efficiency and quality. Over 100

four-phase handshake controller protocols have been verified through ARTIST and Analyze. ARTSIT vastly reduces the design time as compared to hand generation which may take days or even months to achieve a solution set of RT constraints. The quality of ARTIST generated constraints is also shown to be as good as hand generation.

To my family.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Ken Stevens who brought me into the asynchronous world. With his trust, I can continue his favorite research topic on relative timing. Relative timing is the key hub of all the other research in his group. I feel so honored that my work can be applied to others' research. I appreciate all he has done for me, either for work or my family. During this research, I was experiencing the most difficult period I have ever had in my life due to a family emergency. Without his encouragement and care, I could not have made it. This work could not have been done without his guidance, help and patience.

I also would like to thank Marly Roncken, who was the industrial liaison from Intel on SRC project and now is the director of Asynchronous Research Center (ARC) of Portland State University, for her support and help on this relative timing work and instructing me regarding GasP circuit verification. Thank you to Anping He and Professor Xiaoyu Song from Portland State University for the great idea and help in the collaboration on GasP circuit verification.

Next I would like to thank Dr. Chris Myers, Ganesh Gopalakrishnan and Priyank Kalla for their suggestions on related work, and background references. I also would like to thanks Vikas Vij who provided his preliminary results for cycle cutting algorithms to me. Thanks for the funding by grant 1424.001 from Semiconductor Research Corporation (SRC).

Finally I would like to thank my parents, Liangui and Suhua, who have been standing behind me and encouraging me, providing as much as they can do, and especially taking care of my son during the period I am in difficulties. Thanks to my wife, Jingwen, for bringing our son Oscar into this colorful world.

CHAPTER 1

INTRODUCTION

The modern integrated circuit (IC) industry continues to develop extraordinarily fast as predicated by Moore's Law. The number of transistors that can be placed on an integrated circuit doubles approximately every two years. Billions of transistors can be integrated into a single die. The transistors are no longer expensive, and are now almost free.

This miracle depends largely on the use of flip-flops and a clocked synchronous design and verification methodology. This methodology employs a single clock signal as a global timing reference for all the components. Further, the industry standard clocked CAD tools for automating design and verification greatly reduce the time to market. Most engineers focus on register transfer level (RTL) design and verification and are released from complex back-end jobs which are performed mainly by EDA tools.

Design reuse allows heterogeneous IP cores to be integrated on a single system on chip (SoC) to reduce design time. More recently multicore processors are successfully designed and fabricated to increase the capability of parallel computing by multithread programming.

However, there are many problems with today's synchronous circuit design.

- Power consumption. The design consumes more power as the clock periodically switches. Even fine-grained clock gating may not be enough, especially for handheld devices. Modern mobile handset chip manufacturers like Apple, Qualcomm and Broadcom seek low power solutions and finally turn to use low power ARM based architectures.

- Performance. The performance of SoC and multicore processors rely on how efficiently the multiple cores are designed and communicate. The inefficient

design of a switch fabric may degrade the performance of the chip. As the price of transistors goes down, wires become more expensive since they occupy more space, consume more power, and become a major source of delay. This motivates more research on interconnect fabrics and network-on-chip [1].

This dissertation proposes a design methodology targeting 3x improvement on performance and power with asynchronous design over its synchronous counterpart. Timing assumptions of a design usually result in simpler low power and high speed circuits. In this research, relative timing is the key timing methodology employed in an asynchronous design flow using traditional clocked CAD tools not only to guarantee the correctness of the circuits but to drive timing driven synthesis, place and route and postlayout timing validation. Hence generating a correct set of relative timing constraints becomes the key step of this design methodology. This dissertation formally describes the algorithm for automatic generation of relative timing constraints as a replacement of traditional hand generation, which may take an experienced designer days or even months to figure out a complete set of constraints. This algorithm is implemented in a tool called ARTIST (Automatic Relative Timing Identifier based on Signal Traces) and applied to a bunch of asynchronous circuits. The results show that ARTIST can automatically generate a complete set of relative timing constraints in an extremely shorter time while retaining the same quality of constraints compared to traditional hand generation.

1.1 Asynchronous Circuit

Asynchronous circuits are not a new technology, but a resurgence to the semiconductor industry. Asynchronous circuit design has a long history. The research in asynchronous design can be traced back to the mid 1950s [2, 3]. Recently, industry and academia show increased interest in asynchronous design due to power and performance issues as design is getting more complex.

Asynchronous design has the following advantages over synchronous design [4]:

- Low power consumption. Asynchronous design consumes less power than synchronous counterpart because of zero standby power consumption [5, 6].

- High performance. Synchronous design operates at a clock frequency that is determined by the worst-case delay of combinational logic between flip-flops. Asynchronous design, which employs handshaking, operates at actual delay. It is reactive and does not need to wait for a clock edge to proceed.

- No clock distribution and clock skew problems. Asynchronous design employs handshaking protocol for communication instead of the global clock signal.

However, there are drawbacks to asynchronous circuit design that vastly restrict its wide adoption. Unlike synchronous design, asynchronous design lacks uniform CAD tools. Some companies that have succeed in asynchronous circuit design have their own design flow and tools as proprietary properties and not open to the public. Without the aid of tools, asynchronous design still involves much manual work, such as custom layout. This greatly increases the difficulties in asynchronous design. Asynchronous design also requires designers to have experience and expertise in asynchronous circuit design.

The International Technology Roadmap for Semiconductors predicted that 20% of designs will be driven by handshake clocking in 2012, rising to 40% by 2020 [7]. To achieve this target, it is imperative to have some asynchronous design flow that can implement handshake clocking using available clocked CAD tools while requiring little experience of asynchronous design.

1.1.1 Handshake Protocol

Asynchronous design employs handshake protocols instead of using a global reference clock. Communication between asynchronous components is implemented by sending request and receiving acknowledgment signals.

Handshaking protocols can be classified as two-phase and four-phase protocols with respect to a handshake cycle. The four-phase handshaking protocol is implemented by initiating data and asserting request signal. The receiver absorbs the data and asserts acknowledge. The sender de-asserts request upon receiving acknowledge. Finally the receiver de-asserts acknowledge. Another handshake may start if the sender detects that acknowledge is de-asserted. Figure 1.1 shows the transition

relationships of request and acknowledge signals. The handshaking request and acknowledge signals return to zero after one handshaking cycle is finished. The 4-phase handshake protocol is also called return-to-zero (RTZ) signaling or level signaling. The 2-phase handshake protocol shown in Figure 1.2 uses transition signaling instead. The handshaking signals do not return to their initial value after one handshaking cycle is finished. So 2-phase handshake protocol is also called non-return-to-zero (NRZ) signaling or transition signaling.

The 4-phase return-to-zero handshake protocol takes extra transitions to finish a handshake cycle but results in simpler logic implementation. The simple 4-phase circuits can be faster and lower power than 2-phase circuits due to their simplicity. The 2-phase non-return-to-zero handshake protocol theoretically leads to faster designs, but the resulting circuits are more complex.

The handshaking protocol can be implemented completely independent of the data path. This is called a bundled data protocol. The request and acknowledge signals are one bit signals. On the other hand, the request signal can be encoded into data signals. One simple example is the dual rail protocol where the request and one data bit are encoded with two signal wires.

1.2 Timing

Timing is an inherent quality and correctness aspect of circuit and protocol design, whether the designs are clocked or asynchronous. A circuit will not work correctly without functionality and timing correctness. Modern digital circuit design relies heavily on the timing methodology it employs. The following sections describes the synchronous timing and four most often used asynchronous timing methodologies in both industry and academia.

1.2.1 Synchronous Clock

Modern digital IC design favors a synchronous design methodology. In synchronous design, all the components are synchronized by a global clock. It is normally implemented by a employing banks of flip-flops with combinational logic between

them. Flip-flops are edge sensitive storage elements and on every positive or negative edge of clock the flip-flop the input data is sampled.

The clock frequency is determined by the worst delay of the combinational logic between flip-flops. The setup and hold time must be satisfied in order to ensure that the data is correctly latched.

Global clock synchronization and industry standard CAD tools allow engineers to design digital circuits at the behavior level. However, as the design becomes more complex, power, performance and clock distribution become a big issue for synchronous design.

1.2.2 Delay Insensitivity

Delay-insensitive (DI) circuits operate correctly independent of the delay of logic gates and wires. The delay insensitive methodology is the most robust of all asynchronous circuit timing methodologies. However, due to limitations, it is not practical to create delay insensitive systems since it results in larger, slower and power hungry circuits than similar timed circuits [9, 10].

As a practical alternative, quasi-delay-insensitive (QDI) circuits are invariant to the delays of gates and wires, with the exception that certain wires are required to be isochronic forks with identical delays. Of all useful asynchronous design styles, QDI circuits make the fewest timing assumptions, as only the isochronic fork is assumed. There are many successful QDI designs including TITAC from Tokyo Institute of technology [11, 12], MiniMIPS from Caltech [13] and SPA from the University of Manchester [14] among others.

Delay insensitive circuits integrate asynchronous handshaking control logic into data path. All handshaking is implemented with data communication, which is different from a bundled data protocol where the control logic path and data path are separate. A change in sampled data may indicate a start of handshaking. This is implemented by data encoding, normally in the format of a 1-of-n code [15]. Dual-rail encoding is the simplest encoding for delay-insensitive design. It encodes the request signal with the data and uses two wires per data bit for validity or empty.

1.2.3 Metric Timing

Although quasi-delay-insensitive design is tolerant to environmental variation, its conservative timing results in high complexity circuits. Another approach utilizes metric timing constraints to generate timed asynchronous circuits, which result in less circuit complexity.

This approach unfolds the cyclic graph of the specification into an infinite acyclic graph and uses metric timing assumptions to remove the redundancy in the specification and thus results in a finite subgraph for a simpler implementation [16, 17].

The metric timing specifies upper and lower bounds on the delay between signal events becoming enabled and firing [18, 19]. It requires the designer to estimate the min-max delay in a reasonable range such that it meets the accurate delay extracted from postlayout parameters. Further, the impact that a change to the delay of a single component has on the correct behavior of a system as a whole cannot be known by an engineer, making design changes (ECO: Engineering Change Orders) more difficult to perform without re-running the verification.

1.2.4 Unit Delay

The timing of an asynchronous circuits can be analyzed by counting the number of gate delays in a path based on the assumption that all logic gates have the same uniform delay. This is a very straightforward and intuitive way to design and analyze aggressive self-resetting asynchronous circuits such as GasP family circuits [20]. After the circuit is logically verified, the transistors must be properly sized to yield unit delays to meet the assumptions made for correct behavior of the circuit. The method of calculating transistor widths with the aid of logical effort [21] analysis to generate unit delay is presented in [22]. The unit delay model facilitates prelayout timing validation, but the procedure of characterizing transistor sizes is relatively more complex and requires back-end experience and a lot of manual work. However sizing transistor to yield unit delay over-constrains the circuits and degrades their potential performance and power.

1.2.5 Relative Timing

Relative timing is a timing methodology that constrains the firing order of two events based on logic path delays. It fits perfectly into a state based formal verification methodology such that by enforcing relative timing constraints, failure states are made unreachable. Unlike other methods, necessary timing assumptions become explicit when using relative timing. Designers can visualize, reason about, and manipulate path based timing constraints. Enhanced path based relative timing constraints restrict the overall delay of two paths from a common causal point of divergence (POD) to a common point of convergence (POC) to have a specified order of arrival.

One of the advantages of relative timing over other timing methodologies is that path based relative timing constraints can be supported by conventional clocked CAD tools for timing driven synthesis, place and route and pre and postlayout timing validation. A relative timing based design methodology enables synchronous design engineers to switch to asynchronous circuit design using their familiar tools without having too much knowledge of asynchronous circuits.

1.3 Model Checking

Simulation based validation methodologies have been the main stream for validating complex CMOS integrated circuits. However, as design is getting more and more complicated, simulation based validation is not enough to cover all possible scenarios. One cannot enumerate all the possible cases necessary for verification, and some corner cases remain unevaluated. Such a situation is not acceptable, especially for safety critical products. A design must be exhaustively verified. An example of such a failure is the Ariane 5 rocket, which exploded less than 40 seconds after launching.

Model checking is a technique for verifying finite state concurrent systems [23]. Model checking performs an exhaustive reachability analysis of the state space to find any violations of specified properties. Whenever a property is not satisfied, a counter example is returned.

To perform model checking, a design must be modeled in a formal representation which is accepted by the model checker. The specification is a list of properties to be

checked against the design. The process of modeling checking is automatic. When model checking fails, an error trace is returned. This helps the designer to locate and debug the errors.

The properties are normally specified using temporal logics. CTL* formulas describe the properties of computation trees and are composed of path quantifiers and temporal operators. The path quantifiers can be **A** and **E** only where **A** means for *all* computation paths and **E** means for *some* computation path. The temporal operators can be **X** (next time), **F** (finally), **G** (globally), **U** (until) and **R** (release). Temporal logics are often classified into two sublogics, one of which is linear time logic that describes the properties along a single computation path and the other is branching time logic that describes the properties over all the paths that are possible from the current state. An example property specifying the mutual exclusion of two events can be described by temporal logic $\mathbf{G}(\neg e_1 \vee \neg e_2)$.

The main challenge of model checking is state explosion, especially for verifying concurrent systems with lots of concurrency. Symbolic representations for state transition graphs helps mitigate the state explosion problem. Many symbolic representations are based on ordered binary decision diagrams (OBDD) [24]. A BDD represents a boolean formula where each node is a boolean variable and its two outcoming edges denotes if the boolean variable evaluates to true or false. It consists of two terminal nodes called the 0-terminal and 1-terminal. A path from the root to the 1-terminal means that the boolean function is evaluated to be true. The basic idea is from Shannon expansion. The size of a BDD is determined by the ordering chosen for the variables. Finding an optimal ordering of variables is normally not feasible. Hence heuristics are employed for finding a relatively good variable ordering [25, 26]. How to apply formal verification to real world hardware design problems by using PSL [27] or SystemVerilog [28] is described in [29].

1.4 Contributions

The key contributions of this research is automatic formal generation of a complete set of path based relative timing constraints for correctness of circuits and enables clocked CAD flow for asynchronous circuit design.

The algorithm for automatic generation of relative timing constraints vastly reduces design time, which may take days or even months for an experienced asynchronous designer to figure out a complete set of relative timing constraints by hand based on the designer's intuition and expertise on circuit structure and knowledge on asynchronous design. Our one push of button tool ARTIST simply returns a solution set of relative timing constraints and does not require the user to know anything specific to the design. This research may bring up large adoption of asynchronous design by employing clocked CAD tools without expertise in asynchronous circuit knowledge.

As the key step of the asynchronous design flow using conventional clocked CAD tools, the efficiency of constraint generation directly affects the design time of this flow. Without a complete set of relative timing constraints all the subsequent steps by using clocked CAD tools such as timing driven synthesis, place and route and postlayout timing validation cannot be performed.

This work also drives the correct cycle cutting. Inefficient cycle cutting where the relative timing constraints related critical timing paths may be broken results in unexpected power hungry circuits. Given a complete set of relative timing constraints generated from this research work, the synthesis and place and route engines are dictated to remain those relative timing constrained paths intact.

The research work described in this dissertation also has the ability to allow user to specify the desired common timing reference to facilitate postlayout timing validation. Postlayout timing validation is an important step in both clocked and asynchronous design which checks if constrained timing holds with extracted parasitic parameters. To perform timing validation, a virtual clock pin must be specified as a common causal reference to evaluate the delays of two timing paths. This virtual clock pin might be mapped to a primary input, invisible internal or primary output signal. This dissertation supports flexible common causal points since it returns all possible point of divergences. Normally the *request* signal as a primary input signal is mapped into this virtual clock pin. However in case of repartition the circuit hierarchy to facilitate verification such as verifying GasP, an internal signal may be required to

work as the virtual clock signal. User specified point of divergence allows the user to specify desired signal as the common timing reference.

This work also supports unrolling count representation of signal transition where the fall or rise behavior of a transition is modeled using transition counts instead of logic levels. This representation is used for multicycle constraints and especially useful when specifying any relative timing constraint related to clock.

1.5 Dissertation Structure

The dissertation is structured as follows.

Chapter 2 introduces a relative timing based asynchronous design and verification methodology. The relative timing concept is formally defined in Section 2.1. This design methodology allows designers to use traditional clocked CAD tools to design asynchronous circuits. It is implemented by characterizing asynchronous control templates and then mapping the relative timing constraints into sdc constraints such that they are compatible with clocked tools for timing driven synthesis, place and route and pre and postlayout timing validation. A scalable verification method for verifying large compositional asynchronous handshaking protocol using industry symbolic model checking engines is described in Section 2.3.

Chapter 3 describes the formal verification engine employed in this asynchronous design methodology. The formal models for model checking uses the process language Calculus of Communicating System (CCS). The fundamental structure this formal verification relies on and how the formal verification detects internal glitches and check conformance between the implementation and specification are described.

Chapter 4 presents the algorithms for automatic generation of relative timing constraints which is the key work of this thesis. The past work and its weaknesses are described. All types of errors returned from the formal verification engine are formally defined and analyzed. Then the data structure employed for generating relative timing constraints and the key algorithms are described.

Chapter 5 shows a simple C-Element example to demonstrate how the algorithms work on a real example. Another example, 6-4 GasP circuit, demonstrates how

the single track signaling design can be converted to a formal verification engine compatible double track signaling.

Chapter 6 compares the results generated by ARTIST against hand generation in terms of efficiency and quality.

Chapter 7 concludes this dissertation work and addresses possible future work.

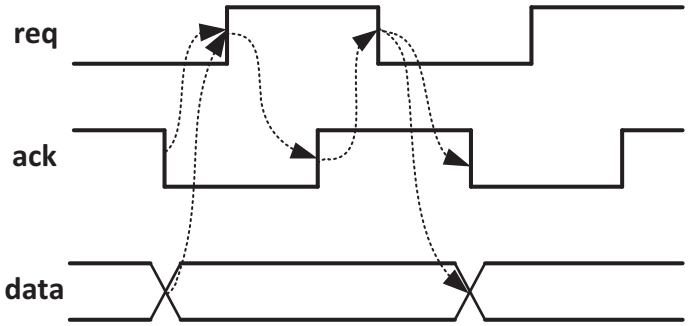


Figure 1.1. Four-phase handshaking protocol.

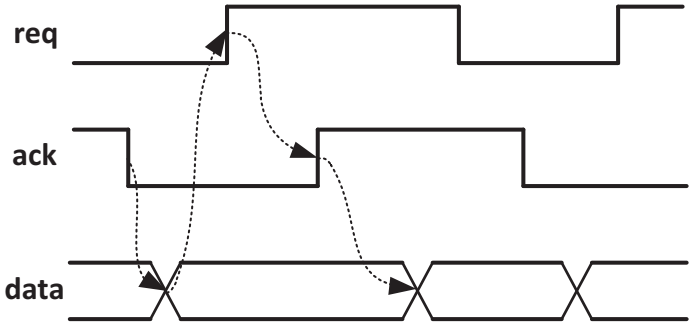


Figure 1.2. Two-phase handshaking protocol.

CHAPTER 2

RELATIVE TIMING BASED DESIGN METHODOLOGY

2.1 Relative Timing

Relative timing is an innovative timing methodology that enables aggressive asynchronous circuit design and verification. It constrains the design by enforcing the firing ordering of two events such that timing failures are made unreachable. Relative timing is applicable to clocked design as well. The setup time constraint of the flip-flop that the data have to be stable at least setup time before the clock edge is triggered is a relative timing constraint as shown in Figure 2.1.

Definition 2.1 *A Relative Timing Constraint specifies a required signal ordering that results from system timing that is imposed between two events that share a common timing reference.*

The behavior of a logic component generally depends on the combinational pattern of input and output values. The state space may be exponential with respect to the number of inputs and outputs. However, in the real operating scenario not all possible sequences will happen. Thus the environment always restricts the behavior of logic components to a subset of the whole. If one can figure out relative timing constraints on inputs that models the environment, the resulting circuit after synthesis can be much simpler than the one that implements the complete set of behaviors [30, 31]. For asynchronous handshake protocol design, relative timing assumptions result in concurrency reduction since the assumption on relative ordering of inputs and outputs makes the protocol more sequential.

Relative timing can be used as a timing constraint for verification. A set of relative timing constraints can make a circuit implementation hazard-free and behave as the specification requires.

When relative timing concept was first proposed, the format of a relative timing consisted only of an ordering of two events such as `a` $\prec$ `b`, which specifies event `a` occurs before event `b`. As a timing assumption that specifies the firing order of primary inputs, this format is enough to represent the environment behavior. However specifying the relative ordering of two events for verification is not enough because this format cannot be supported by timing analysis engines for postlayout timing validation. The enhanced format of relative timing adds a point-of-divergence (POD) onto the relative ordering to form path-based timing constraints that are able to be validated in postlayout static timing analysis engines. A path-based relative timing constraint is represented as POD $\mapsto$ POC_0 $\prec$ POC_1. Figure 2.2 interprets the meaning of this representation - the delay of the path from POD to POC_0 is less than the delay of the path from POD to POC_1, i.e., $\triangle_{POD-POC_0} < \triangle_{POD-POC_1}$. Block POD and POC represents logic gates and block A and B represents either logic gates or just wires.

Relative timing is more straightforward, especially when the system is modeled as a state transition graph. Given a particular state that has concurrent transitions (have two or more egress transitions whereas a state that has only one egress transition is deterministically sequential), a relative timing constraint enforces a design to always choose the path with a smaller delay. The subgraph directed from the longer path will be never reachable. Figure 2.3 illustrates how the relative timing constraint impacts the state transition graph. If the relative ordering `b`+ $\prec$ `a`$-$ is applied to the partial graph in Figure 2.3 and then the subgraph in dashed line is no longer reachable. Note that the relative timing constraint is not truncating a graph but makes the partial graph unreachable.

2.2 Asynchronous Design Flow Using Clocked CAD Tools

Asynchronous circuits, albeit impressive in power and performance benefits compared to clocked circuits, is not widely adopted mainly because of the lack of supporting CAD tools and requiring deep expertise in asynchronous circuit design knowledge.

Rather than compete in the CAD domain and develop distinctly independent design flows, a relative timing based design methodology is proposed that exploits traditional commercial CAD tools to facilitate asynchronous circuit design and verification [32]. The design flow is shown in Figure 2.4. This design and verification methodology allows the designer to apply commercial clocked CAD as much as possible. This approach consists of two major procedures: asynchronous template characterization and traditional system design that employs precharacterized templates.

Once the asynchronous templates are fully characterized, synchronous designers can use them as library cells to build a system by following clocked design flow. This enables designers who have been working on synchronous design to switch to asynchronous circuit design smoothly with little expertise in the asynchronous domain.

This design flow applies to all kinds of asynchronous circuit designs, including desynchronization. Desynchronization is a process of converting synchronous circuit into an asynchronous one [33, 34, 35]. To desynchronize a synchronous design, its clock tree is replaced with handshake controllers, but the combinational logic between registers in the data path remains untouched. This replacement perfectly fits the bundled data protocol, which separates the control path and data path.

A simple example of desynchronization shown in Figure 2.5 will be used to demonstrate the asynchronous design flow in detail. It is a pipelined design that implements the function of $x^2 + 3x$ where a 16-bit wide data path forks x out to upper and lower data paths performing multiplications concurrently and then joining the paths to perform an addition operation. The control path is composed of linear controllers (LC) and fork join (F/J) modules. This linear controller implements a four-cycle return-to-zero handshake protocol. This is a timed protocol and follows the burst mode assumption that assumes that the circuit stabilizes before any new inputs can

be accepted [36, 37]. The data path is composed of registers (R), either flip-flops or latches. The oval boxes represent arithmetic operations. The top level Verilog code for a latch based implementation is shown in Figure 2.6.

2.2.1 Formal Verification of Asynchronous Templates

Formal verification and relative timing constraint generation of asynchronous templates are the key steps of the design flow. Templates refer to the local asynchronous controllers that can be instantiated one or multiple times for building a system. Formal verification is the process of creating a complete set of relative timing constraints that guarantee the correctness of a template.

The asynchronous templates in the example are the linear controller and the fork-join modules. The circuit diagram of the linear controller is shown in Figure 2.7 and its Verilog representation is shown in Figure 2.8.

The template is formally verified in an untimed manner that assumes unbounded delay on both gates and wires by a bisimulation relation based formal verification engine. Thus the circuit implementation and specification are required to be modeled with formal representations which can be recognized by the formal verification engine. The Calculus of Communicating System (CCS) [38] is selected as the process language for the formal model because it formally supports verification of nondeterminism such as arbiters and synchronizers by its distinct support for invisible internal τ transitions. The CCS specification of the linear controller is shown in Figure 2.9. A Verilog netlist can be converted into a formal CCS model automatically by a tool called *verilog2ccs* which is the V2CCS block shown in Figure 2.4. This tool takes three input files and outputs the CCS implementation of the circuit.

- A structural Verilog file that consists of primitive gates as the implementation of the template. See Figure 2.8.
- A mapping file of Verilog gates to formal semi-modular description of each gate in CCS. See Figure 2.10.
- A functional description of the gates in the target technology. See Table 2.1.

The converted CCS implementation of the linear controller is shown in Figure 2.11. The tool also has the ability to calculate the initial semi-modular state of each gate, i.e., the initial value of inputs and outputs of each gate. For example, A121O2I0bc01 defines initial values of `lr`, `ra_`, `y_`, `la` and `la_` to be 0, 1, 1, 0 and 1 respectively.

Untimed model checking is then performed between the circuit implementation and specification, which is the RT-FV block shown in Figure 2.4. The first run of formal verification performs speed-independent verification that assumes unbounded delay on gates and zero delay on wires. This generally results in numerous violations, many of which are due to technology mapping. These violations may cause internal glitches that may finally propagates to the primary outputs and result in failure of the design. Relative timing constraints must be generated to remove these violations by restricting the reachability of failure states with circuit timing. By applying the relative timing constraints to the implementation recursively, the design conforms to the specification. The set of relative timing constraints created during speed-independent verification produces the key set of timing constraints for timing driven sizing and place and route. The set of relative timing constraints for the speed-independent run on this linear controller is shown in the SI rows of Table 2.2.

The second run of formal verification is verification of the timing properties of the template protocol. Some protocols are timed protocols that may not accept all signal behaviors of its environment. Protocol verification is performed to verify that the interaction between local templates is correctly performed. For a linear pipeline, three of the same templates can be composed in series as shown in Figure 2.12. For other generic asynchronous systems, protocol verification at the system level requires the templates to be composed as specified. Instead of using the plain specification of templates, protocol verification requires minimized specification which can be generated from Concurrency Work Bench (CWB) [39] with the *min* command. The minimized specification of the linear controller is shown in Figure 2.13. The set of relative timing constraints generated for the second run are key constraints for timing driven sizing and place and route as well which is shown in the Protocol row of Table 2.2.

A third verification run is performed to generate any timing constraints between the handshake clocking and the datapath logic. Like clocked design, handshake clocking follows the same setup constraint – data has to be stable at least some setup time before the relevant handshaking signal is triggered, e.g., $\mathrm{lr}\uparrow \mapsto \mathrm{din} \prec \mathrm{la}\uparrow$. When a design that employs the bundled data protocol is synthesized, such constraints create a matched delay between the datapath and control logic. This is guaranteed by constraining that the minimum relative delay of the control path to be larger than the maximum delay of data path. This set of relative timing constraints is key for timing driven synthesis and place and route of creating matched delay in the pipeline.

The final verification run performs delay insensitive verification, which not only assumes unbounded delay on gates but also wires. The wire fork is not isochronic any more. It is modeled with an unbounded delay in an arbitrary order on the two branching wires. Delay insensitive verification is necessary for some asynchronous circuits that makes use of wire delays to achieve extremely aggressive timing to maximize throughput, such as GasP family circuits. The set of relative timing constraints for delay-insensitive verification is shown in the DI row of Figure 2.2.

A template is fully characterized with a complete set of relative timing constraints generated by the above four rounds of verification runs. The process of generating relative timing constraints can be manually done based on designer's strong knowledge of asynchronous circuits and his/her understanding of the circuit structure of the design under test. It is quite time-consuming and prone to errors. Generating a complete set of relative timing constraints for a design may take an experienced designer hours or even days. The objective of this dissertation is to present a method that can automatically generate relative timing constraints that are a key part of this design flow.

2.2.2 Template Characterization

After a complete set of relative timing constraints is derived, the design enters RT flow phase where template characterization and mapping relative timing constraints to backend are performed.

The set of relative timing constraints for template characterization is required to be mapped into compatible sdc constraints such that they can be supported by conventional CAD tools for timing driven synthesis, place and route and postlayout timing validation.

Synopsys tools support setup and hold constraint checking between two *data* signals where neither of them is a clock signal. This is implemented with set_data_check command. But its fundamental principle is similar to clock based setup and hold checks which assume one of the *data* signals is considered as a clock pin, called the related pin, while the other is regarded as traditional data, called the constrained pin. A data check example is shown in Figure 2.14. The related pin D2 is regarded as a reference clock pin and the constrained data is checked for setup and hold violation according to the reference. The set_data_check command takes a value that specifies a setup or hold time for which D1 must be stable before or after D2 goes high. The options of command set_data_check is shown below.

command **set_data_check**
race_margin
-clock
-from | -rise_from | -fall_from *related_pin*
-through *traverse_pin*
-to | -rise_to | -fall_to *constrained_pin*
-setup | -hold

Since CAD is designed for clocked design, a -clock argument is always an option of this command as a common reference point. The -clock option in set_data_check specifies the starting point to related and constrained pins such that the delays of the two paths are qualified for comparison. Asynchronous design has no clock signal and thus a virtual clock signal must be specified. The point-of-divergence of path based relative timing constraints exactly fits the -clock option. In asynchronous design, this virtual clock pin is normally mapped to a request signal. The related and constrained timing paths can be specified using the -from and -to options. The shorter path of a relative timing constraint uses the -from option, and longer path uses -to option followed by the specific pin names. More concretely, the transition behaviors of the two racing events of relative timing constraints can be modeled by -rise_from,

-fall_from, -rise_to and -fall_to options. Generally there is more than one path available for evaluation, and the CAD tool may not report the exact one wanted. In such a case the -through option is used to specify the pin points the desired path passes through. The options -setup and -hold are mutually exclusive. Only one of them will appear in the single sdc constraint. The command set_data_check is mostly used for postlayout timing validation.

The sets of relative timing constraints in speed-independent, protocol and delay-insensitive verifications are all mapped into set_data_check constraints, which is shown in Table 2.3.

The storage elements in the data path, using either flip-flops or latches, still need to obey setup and hold constraints. If combinational logic exists between pipeline stages for data processing, the processed data normally takes more time to propagate to the storage element of the next stage. On the other hand, in the control path pipeline, handshaking is performed much faster than the data path and thus the signal ordering and setup time cannot be guaranteed. Hence delay elements must be added into the control path to match data path delay such that the data is guaranteed to be available when handshake clocking arrives. This is implemented by a pair of commands – set_max_delay and set_min_delay as shown below.

command **set_min/max_delay**
delay_value
-from | -rise_from | -fall_from *start_pin*
-to | -rise_to | -fall_to *end_pin*

Set_max_delay command is used to constrain the data path while set_min_delay command is used to constrain the control path. The delay from the output of previous stage data storage element to the input of next stage storage element is constrained by set_max_delay by a delay value such that the delay of combinational logic between them must have a maximum delay of that value. Likewise the set_min_delay constrains the control path to have the minimal given delay value. Since both commands specify end-to-end delay, option -from and -to are enough to denote the starting and end pins. The following example set_min/max_delay constraints constrain the maximum delay from register bank R0 to R10 to be 1.7ns and the minimum delay from the linear

control associated with R0 to the linear control associated with R10 to be 1.7ns as well. This guarantees that the data always arrive before the control signal.

```
set_max_delay 1.7 -from [get_pins R0_reg_latch*/Q] \
                  -to   [get_pins R10_reg_latch*/D]
set_min_delay 1.7 -rise_from [get_clocks tk0/lr] \
                  -rise_to   [get_pins tk10_lc1/A0]
```

The synthesis and place and route tools may automatically optimize circuits, such as merging back-to-back inverters, combining multiple simple primitive gates into a complex gate or vice versa. This modification breaks the original structure and characteristics of the asynchronous templates and introduces unexpected timing hazards. The command set_size_only prevents the logic structure of the templates from being modified by the CAD tools, and only allows the tools to optimize the drive strength of the gates to gain better power and performance. Another command, set_dont_touch, disallows the tool from modifying the design in any manner. The hierarchical components of templates should be constrained by one of these two commands. The following constraint disallows any structural modification on the AOI gate of linear controller.

```
set_size_only -all_instances { */lc3 }
```

Clocked CAD tools operate on directed acyclic graphs (DAGs) for timing driven optimization. Once a cycle is found in timing graphs, the CAD tools will invoke built-in algorithms to break the cycle. The users can also define how and where to break the cycle using set_disable_timing by themselves. Asynchronous sequential circuits inherently have cycles in the design due to its sequential characteristics. The handshake protocols themselves also produce cycles. These cycles must be cut to be compatible with the CAD tools. The built-in cycle cutting algorithm of clocked CAD tools may be good enough for clocked design. However, the timing driven synthesis and place and route require the relative timing constraints to be successfully applied to the design and need all relative timing constraints related timing paths to remain unbroken. This requires that the paths from point-of-divergence to point-of-converge

of the relative timing constraints are forbidden to being cut. Hence custom cycle cutting algorithms are necessary. An algorithm for automatic cycle cutting, as part of this asynchronous design flow, is being developed. The set_disable_timing constraint is applied to primitive gates and the timing arc is removed from the specified input pin (-from option) to the specified output pin (-to option). In this example, both local cycles and handshake cycles are cut as shown in Table 2.4.

The set of relative timing constraints from the speed-independent run and min/max constraints are key constraints for timing driven sizing and place and route. The set of constraints for protocol verification do not usually need to be included in synthesis and place and route because of the magnitude of slack between the two race paths. The set of relative timing constraints from the delay-insensitive run is not used for synthesis but used for postlayout timing validation.

2.2.3 Mapping to Backend

The relative timing constraints must be mapped to backend format of constraints with full hierarchical path names.

An enhanced format of sdc constraint allows the mixed use of module names and instance names in defining hierarchical port names [40]. Variables are also supported to be specified in hierarchical port names to reduce tediously duplicating constraints for each instantiated templates. This is used to map timing constraints generated for an asynchronous design template into its instances used in a design.

2.2.4 Postlayout Timing Validation

Timing validation using standard static timing analysis engines is employed to guarantee that the constrained timing holds with extracted parasitic parameters. All the relative timing constraints that are either applied to timing driven synthesis and place and route as well as the delay-insensitive constraints are required for performing postlayout timing. The report_timing command is used to return a detailed timing report for each constraint by listing all the nodes the path passes through and their corresponding delays. The necessary constraint settings for the relative timing constraint $\texttt{lr}+ \Rightarrow \texttt{rr}+ \prec \texttt{y}-$ are shown below. Figure 2.15 shows the related timing

report. The timing report lists details the delay information of the two paths from the point-of-divergence to the point-of-convergence and compares the total delay to see if constrained timing holds.

```
create_clock [list [get_pins tk0_lc1/A0]
                   [get_pins tk0_lc1/B0]
                   [get_pins tk0_lc3/A1]]
             -name tk0/lr  -period 1.7  -waveform {0 0.85}
set_data_check -clock [get_clocks tk0/lr]
               -fall_from [get_pins tk0_lc3/A2]
               -rise_to [get_pins tk0_lc3/B1]
               -setup 0.05
```

2.3 Verifying Compositional Asynchronous Protocols

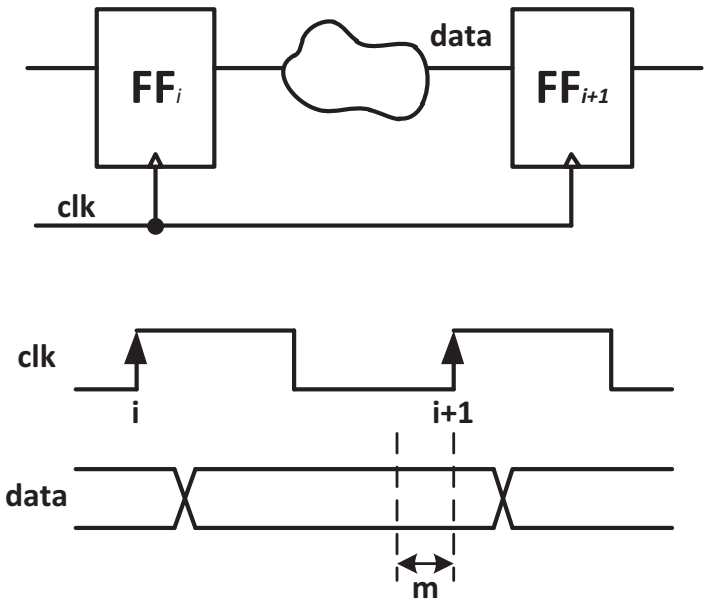
This system level design methodology incorporates the composition of multiple precharacterized asynchronous handshake protocols. System level verification is employed to check any violations in the communication of these protocols. Each protocol may be a timed protocol, which must be constrained to be compatible with its adjacent environmental behavior. The timing required to specify environmentally friendly behaviors is implemented by relative timing constraints.

System level formal verification is perform to guarantee the correct interactions of local protocols. The state explosion problem has been a primary challenge of formal model checking specially for asynchronous circuits and protocols where much concurrency exists. The explicit state based formal verification engine such as *Analyze* may not be applicable to relatively large and complex design.

A scalable verification methodology for compositional asynchronous hardware protocols uses mature symbolic model checking engines [41] to mitigate the state explosion issue during verification [42]. First, the a state graph based representation of the protocols is upgraded to an extended state graph with their timed relative timing property constraint information. The relative timing constraints are represented by making use of a relative timing variable where the variable is set when the point-of divergence fires and reset when the shorter path point-of-convergence signal transition

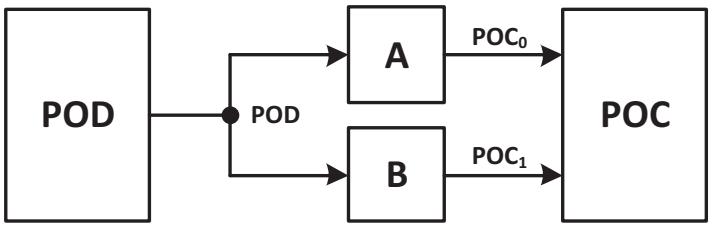
fires. The longer path point-of-convergence signal transition can fire only after the variable is reset. Hence the formal model of protocol and corresponding relative timing constraint is derived from this extended state graph. Second, properties such as safety, liveness, and semimodularity are generated. Finally symbolic model checking is performed by the industry symbolic engine NuSMV. If the properties specified are satisfied, the composed protocols can interact correctly. If this fails, a counter example is reported and further investigation must be performed to guarantee if missing relative timing constraints exist.

This methodology allows us to verify larger designs that are composed of heterogeneous timed asynchronous handshake protocols using relative timing and symbolic model checking techniques more efficiently.



$$clk_i \mapsto data \prec clk_{i+1} + m$$

Figure 2.1. Relative timing application to clocked system.



$$\triangle_{POD-POC_0} < \triangle_{POD-POC_1}$$

Figure 2.2. Circuit diagram to demonstrate path-based relative timing constraint.

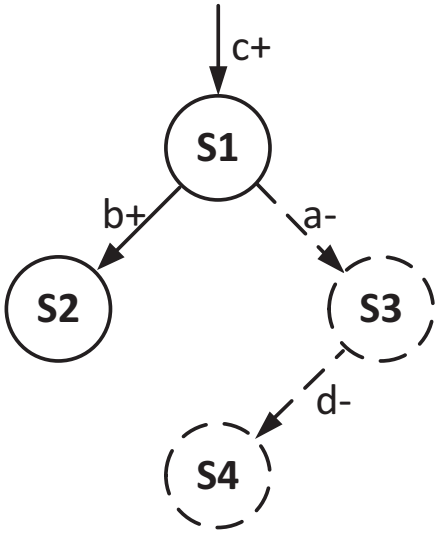


Figure 2.3. Applying $\mathtt{b}+ \prec \mathtt{a}-$ to state transition graph.

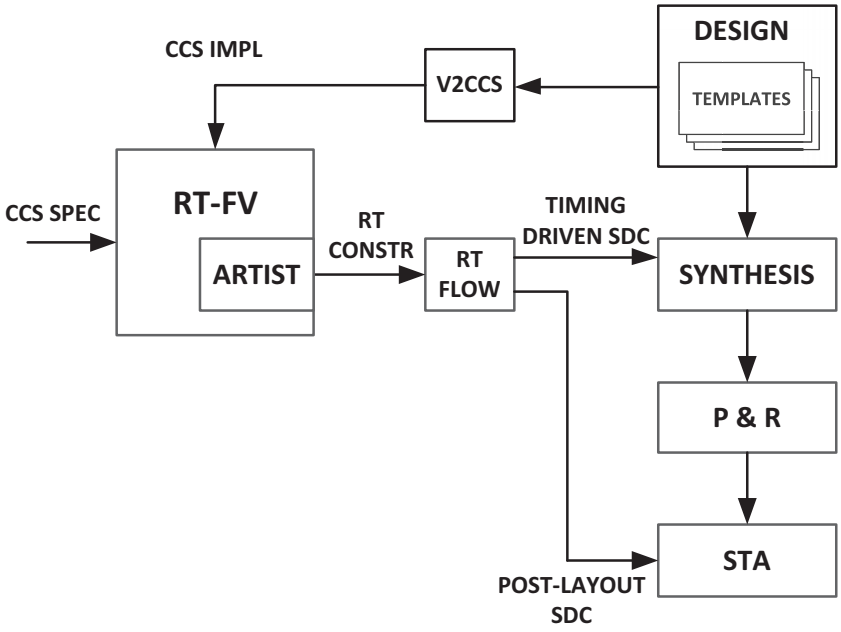


Figure 2.4. Relative timing based asynchronous design flow.

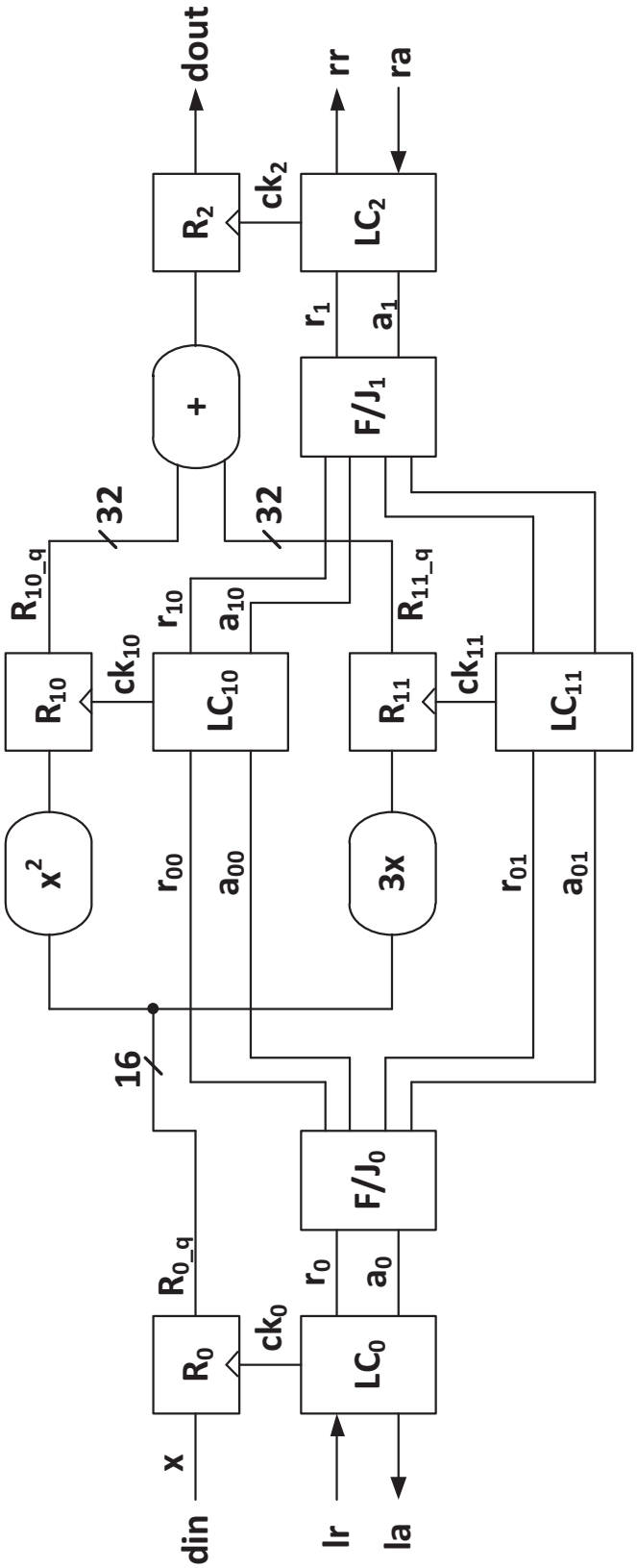


Figure 2.5. Example design: a simple ASIC mathematical pipeline segment computing out $= x^2 + 3x$.

```
module apipeline (din, dout, lr, la, rr, ra, rst);
   input lr, ra, rst;
   output la, rr;
   input [15:0] din;
   output [31:0] dout;
   reg [31:0] R0, R10, R11, R2;
   ...
   assign dout = R2_q;

   always @(*)    R0 = din;
   linear_control       lc0       (.ck(ck0), .lr(lr), .la(la), .rr(r0), .ra(a0), .rst(rst));
   latch_active_high    R0_reg    (.d(R0), .clk( ck0), .q(R0_q));
   bcast_fork           bcf0      (.bi(r0),.bo0(r00),.bo1(r01),.ji0(a00),.ji1(a0 1),.jo(a0));
   always @(*)    R10 = R0_q * R0_q;
   linear_control       lc10      (.ck(ck10), .lr(r00),.la(a00),.rr(r10),.ra(a10 ),.rst(rst));
   latch_active_high    R10_reg   (.d(R10), .clk( ck10), .q(R10_q));
   always @(*)    R11 = R0_q * 3;
   linear_control       lc11      (.ck(ck11), .lr(r01),.la(a01),.rr(r11),.ra(a11 ),.rst(rst));
   latch_active_high    R11_reg   (.d(R11), .clk( ck11), .q(R11_q));
   bcast_fork           bcm0      (.bi(a1),.bo0(a10),.bo1(a11),.ji0(r10),.ji1(r1 1),.jo(r1));
   always @(*)    R2 = R10_q + R11_q;
   linear_control       lc2       (.ck(ck2), .lr(r1), .la(a1), .rr(rr), .ra(ra), .rst(rst));
   latch_active_high    R2_reg    (.d(R2), .clk( ck2), .q(R2_q));
endmodule // apipeline
```

Figure 2.6. Top level Verilog for latch based implementation example.

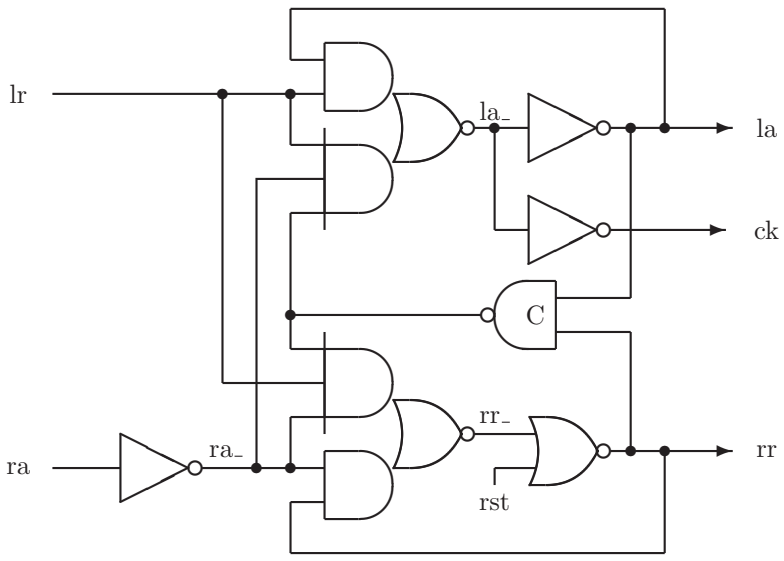


Figure 2.7. LC circuit implementation.

```
module linear_control (lr, la, rr, ra, ck, rst);
   input              lr, ra, rst;
   output             la, rr, ck;
   AOI32X2A12TH   lc0   (.A1(lr), .A2(ra_), .A3(y_), .B1(lr), .B2(la), .Y(la_ ));
   AOI32X2A12TH   lc1   (.A1(lr), .A2(ra_), .A3(y_), .B1(ra_), .B2(rr), .Y(rr _));
   NOR2X2A12TH    lc2   (.A1(la), .A2(rr), .Y(y_));
   INVX2A12TH     lc3   (.A1(la_), .Y(la));
   INVX2A12TH     lc4   (.A1(la_), .Y(ck));
   NOR2X2A12TH    lc5   (.A1(rst), .A2(rr_), .Y(rr));
   INVX2A12TH     lc6   (.I(ra), .Y(ra_));
endmodule // linear_control
```

Figure 2.8. Verilog implementation of linear controller.

```
L     =  lr.c1.'la. c2.lr.'la.   L
R     =    'c1.'rr.'c2.ra.'rr.ra.R
SPEC  =  (L | R) \ {c1, c2}
```

Figure 2.9. CCS specification of linear controller.

```
module artisan65nm2ccs ();
   NAND3X2A12TH      NAND0001      (.A(a), .B(b), .C(c) , .Y(d));
   NOR2X2A12TH       NOR001        (.A(a), .B(b), .Y(c) );
   AOI2XB1X2A12TH    A2B1O2I0001   (.A0(b), .A1N(a), .B 0(c), .Y(d));
   OAI21X2A12TH      O12A2I0001    (.A0(b), .A1(c), .B0 (a), .Y(d));
endmodule // artisan65nm2ccs
```

Figure 2.10. Gate library to CCS specification mapping.

```
LC-IMPL =
(  A121O2I0bc01[lr/a, ra_/b, y_/c, la/d, la_/e]      \
 | INV[la_/a, la/b]                                  \
 | A121O2Ia0c01[ra_/a, lr/b, y_/c, rr/d, rr_/e]      \
 | INV[rr_/a, rr/b]                                  \
 | NOR001[la/a, rr/b, y_/c]                          \
 | INV[ra/a, ra_/b]                                  \
) \ { y_, la_, rr_, ra_ }
```

Figure 2.11. CCS implementation of linear controller.

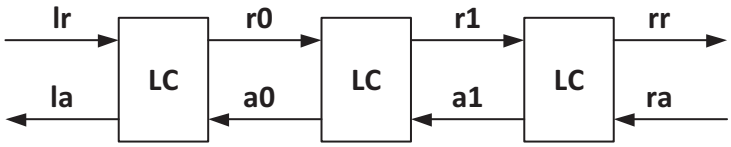


Figure 2.12. Three deep pipeline of linear controller.

```
SPEC*P0M0M0  =   lr.SPEC*P0M0M1
SPEC*P0M0M1  =  'rr.SPEC*P0M0M2  + 'la.SPEC*P0M0M3
SPEC*P0M0M3  =  'rr.SPEC*P0M0M4
SPEC*P0M0M4  =   lr.SPEC*P0M0M5  +  ra.SPEC*P0M0M6
SPEC*P0M0M6  =  'rr.SPEC*P0M0M10 +  lr.SPEC*P0M0M13
SPEC*P0M0M13 =  'rr.SPEC*P0M0M14 + 'la.SPEC*P0M0M12
SPEC*P0M0M12 =  'rr.SPEC*P0M0M11 +  lr.SPEC*P0M0M17
SPEC*P0M0M17 =  'rr.SPEC*P0M0M8
SPEC*P0M0M8  =   ra.SPEC*P0M0M1
SPEC*P0M0M11 =   lr.SPEC*P0M0M8  +  ra.SPEC*P0M0M0
SPEC*P0M0M14 =  'la.SPEC*P0M0M11 +  ra.SPEC*P0M0M16
SPEC*P0M0M16 =  'la.SPEC*P0M0M0
SPEC*P0M0M10 =   lr.SPEC*P0M0M14 +  ra.SPEC*P0M0M15
SPEC*P0M0M15 =   lr.SPEC*P0M0M16
SPEC*P0M0M5  =  'la.SPEC*P0M0M9  +  ra.SPEC*P0M0M13
SPEC*P0M0M9  =   lr.SPEC*P0M0M7  +  ra.SPEC*P0M0M12
SPEC*P0M0M7  =   ra.SPEC*P0M0M17
SPEC*P0M0M2  =  'la.SPEC*P0M0M4
```

Figure 2.13. Minimized specification of linear controller.

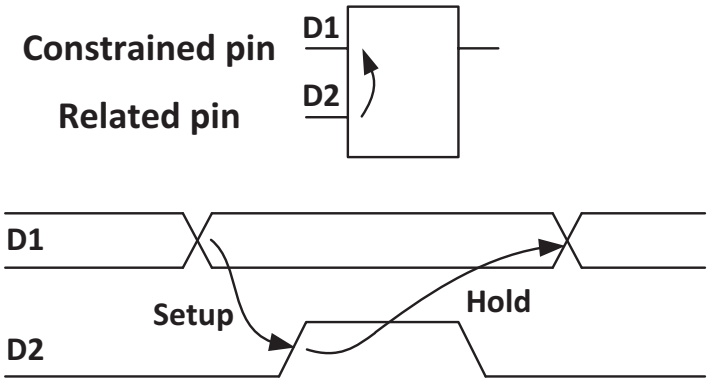


Figure 2.14. An example of data check.

```
Startpoint: tk0_lc3/A1 (clock source 'tk0/lr')
Endpoint: tk0_lc3 (falling edge-triggered data to data check
          clocked by tk0/lr)
Path Group: tk0/lr
Path Type: max

Point                                       Incr       Path
-----------------------------------------------------------
clock tk0/lr (rise edge)                    0.00       0.00
clock source latency                        0.00       0.00
tk0_lc3/A1 (AOI32X1A12TH)                   0.00       0.00 r
tk0_lc3/Y (AOI32X1A12TH)                    0.27 *     0.27 f
tk0_lc4/Y (NOR2X8A12TH)                     0.11 *     0.38 r
U243/ECK (FRICGX0P5BA12TH)                  0.15 *     0.53 r
U244/Y (BUFHX1P4A12TH)                      0.07 *     0.60 r
U245/Y (DLY2X0P5A12TH)                      0.14 *     0.74 r
U246/Y (DLY4X0P5A12TH)                      0.58 *     1.32 r
tk0_lc3/B1 (AOI32X1A12TH)                   0.00 *     1.32 r
data arrival time                                      1.32

clock tk0/lr (rise edge)                    0.00       0.00
clock source latency                        0.00       0.00
tk0_lc3/A1 (AOI32X1A12TH)                   0.00       0.00 r
tk0_lc3/Y (AOI32X1A12TH)                    0.26 *     0.26 f
tk0_lc4/Y (NOR2X8A12TH)                     0.11 *     0.37 r
U243/ECK (FRICGX0P5BA12TH)                  0.15 *     0.52 r
U244/Y (BUFHX1P4A12TH)                      0.07 *     0.60 r
U245/Y (DLY2X0P5A12TH)                      0.14 *     0.74 r
U246/Y (DLY4X0P5A12TH)                      0.58 *     1.31 r
tk0_lc5_c_element2/Y (NAND2X1A12TH)         0.13 *     1.44 f
tk0_lc5_c_element3/Y (NAND3X1A12TH)         0.10 *     1.54 r
U1130/Y (INVX2A12TH)                        0.05 *     1.59 f
tk0_lc3/A2 (AOI32X1A12TH)                   0.00 *     1.59 f
data check setup time                      -0.05       1.54
data required time                                     1.54
-----------------------------------------------------------
data required time                                     1.54
data arrival time                                     -1.32
-----------------------------------------------------------
slack (MET)                                            0.22
```

Figure 2.15. Timing report of constraint lr+ ⇒ rr+ ≺ y−.

Table 2.1. CCS specification functional descriptions.

CCS Cell Name	**SigIndex**	**Output**	**Function**
function NAND0001	4	d	not(a * b * c)
function NOR001	3	c	not (a + b)
function A2B1O2I0001	7	d	not((not(a)*b) + c)
function O12A2I0001	6	d	not(a * (b + c))

Table 2.2. RT constraints for linear controller.

Category	RT Constraints
SI	lr+ ⇒ y_− ≺ la− lr+ ⇒ y_− ≺ rr−
Protocol	lr+ ⇒ ra_− ≺ la+ lr+ ⇒ lr− ≺ rr+ lr+ ⇒ la− ≺ y_− lr+ ⇒ rr− ≺ y_−
DI	lr+ ⇒ y_+ ≺ lr− lr+ ⇒ y_+ ≺ ra_− lr+ ⇒ rr_− ≺ rr−

Table 2.3. Set_data_check constraints of linear controller.

Category	RT Constraints
SI	set_data_check -fall_from */lc1/A2 -fall_to */lc1/B1 -setup $race_margin set_data_check -fall_from */lc3/A2 -fall_to */lc3/B1 -setup $race_margin
Protocol	set_data_check -fall_from */lc1/A1 -rise_to */lc1/B1 -setup 0 set_data_check -fall_from */lc3/A1 -rise_to */lc3/B1 -setup 0 set_data_check -fall_from */lc5/A -rise_to */lc5/Y -setup 0 set_data_check -fall_from */lc5/B -rise_to */lc5/Y -setup 0
DI	set_data_check -rise_from */lc3/A2 -fall_to */lc3/A1 -setup 0 set_data_check -rise_from */lc1/A2 -fall_to */lc1/A1 -setup 0 set_data_check -fall_from */lc4/A -fall_to */lc4/Y -setup 0

Table 2.4. Cycle cutting constraints.

Category	Constraints
Local	set_disable_timing -from A2 -to Y [find -hier cell *lc1] set_disable_timing -from B1 -to Y [find -hier cell *lc1] set_disable_timing -from A2 -to Y [find -hier cell *lc3] set_disable_timing -from B1 -to Y [find -hier cell *lc3]
Handshake	set_disable_timing -from A1 -to Y [find -hier cell *lc1] set_disable_timing -from A1 -to Y [find -hier cell *lc3] set_disable_timing -from B0 -to Y [find -hier cell *lc3]

CHAPTER 3

FORMAL VERIFICATION ENGINE

The formal verification engine employed in this design flow is an explicit state verification engine [43]. It is an untimed verification engine that does reachability analysis using all possible delay scenarios. The verification engine takes an implementation I, optionally a specification S, and a set of relative timing constraints C which is initially empty and outputs an error trace when there is a violation of a semimodular constraint or nonconformance between the implementation and the specification.

3.1 Modeling Concurrent System Using CCS

To use formal tools, both the specification and circuit implementation need to be modeled in a modeling language specific to the formal tool. There are many good modeling languages that are widely used today such as CSP [44, 45] and Petri-net [46]. The formal verification engine used for this research uses Calculus of Communication System (CCS) as the modeling language.

CCS is powerful for modeling concurrent systems. CCS can model very complex parallel systems only using five constructions and six transition rules. CCS syntax does not distinguish logic levels of signal transitions. Thus the state space used for modeling a system can be less than a traditional one where logic levels are specified. Figure 3.1 shows a comparison between a CCS model and a traditional model of a C-element. The CCS model in Figure 3.1(a) has four states, whereas the traditional model in Figure 3.1(b) has eight states. Therefore CCS modeling always results in half the state space as a traditional formal model for this design. In addition, CCS has the ability to model hierarchy. Local blocks can be modeled separately as CCS agents that are composed into a higher level design. It also supports silent internal actions

that make autonomous communications between agents. CCS is rich in equational reasoning as well.

CCS syntax contains five constructions.

- **Prefix** specifies sequential behavior between two events, or an event followed by a process, using the prefix operation ".". For example, $\alpha.\beta$ means that if event α occurs it must be followed by the event β. $\alpha.P$ means that once event α occurs then process P is true.

- **Summation** implements nondeterministic choice with the "+" operator. For example, $\alpha.P + \beta.Q$ represents that if α occurs process P is true whereas if β occurs process Q is true. The firing of α and β is completely non-deterministic.

- **Parallel Composition** allows the composition of local agents with the "|" operator. The composed agents evolve concurrently. For example, if transition τ is an output of agent P and input of agent Q, then the system evolves as $P|Q \xrightarrow{\tau} P'|Q'$.

- **Restriction** limits the scope of a signal to be local to the current agent. This construction is composed of the "\" operator followed by a list of internal signals in curly braces.

- **Relabeling** renames the signals in an agent with the format of "$NewName/OldName$" using the "/" operator.

3.2 Labeled Transition System

The verification engine used in this thesis is an explicit state verification engine built on a labeled transition system. The labeled transition system and related notations are defined as follows:

Definition 3.1 *A labeled transition system,* $(S, T, \{\xrightarrow{t} : t \in T\})$ *consists of*

- *a set* S *of states*

- *a set* T *of transition labels*

- *a transition relation* $\xrightarrow{t} \subseteq S \times S$ *for each* $t \in T$

Definition 3.2 *The labels (or actions) in labeled transition systems are defined as follows:*

- *Input action set* **names** $a \in \mathcal{A}$ *(the set of names* $\mathcal{A}$ *are inputs* $\mathcal{I}$*).*
- *Output action set* **conames** $\overline{a} \in \overline{\mathcal{A}}$ *(the set of conames* $\overline{\mathcal{A}}$ *are outputs* $\mathcal{O}$*).*
- *The set of* **labels** $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$*.*
- *The invisible internal action* τ *(tau).* $\tau \notin \mathcal{L}$*.*
- *The actions of a system are:* $Act = \mathcal{L} \cup \{\tau\}$*.*
- *The* **sort***(P) of an agent P is its complete set of observable input and output actions.*

The set of labels $\mathcal{L}$ consists of the set of primary inputs $\mathcal{A}$ and the set of primary outputs $\overline{\mathcal{A}}$ of a system. The communications within a system are performed by internal silent transitions τ. If the system only contains a primitive gate where τ is empty, the *Act* only contains inputs $\mathcal{A}$ and outputs $\overline{\mathcal{A}}$. Within a hierarchical system, the output signals of one element and its receiving elements follow the convention that **names** represent input signals and **conames** represent output signals (**labels** and **colabels** are the alternative names). These signals become internally abstracted as the invisible internal action τ. To distinguish the difference between the internal interactions of different signals, the **labels** and **colabels** of each internal transition τ can be denoted specifically as $\tau(\alpha)$ and $\tau(\overline{\alpha})$. Figure 3.2 demonstrates how the internal transition $\tau(a)$ interacts between two agents P and Q.

Definition 3.3 *If* $s \in Act^*$ *is an action sequence of an agent, then* $\hat{s}$ *is defined to be the* projection *of* s *on* $\mathcal{L}^*$*, i.e.* $\hat{s}$ *is the sequence obtained from* s *by deleting all occurrences of* τ*.*

The transition relation symbol $\Rightarrow$ is used to represent an action sequence where invisible internal τ transitions can be concatenated with an action α or sequence s.

It is generally used together with $\hat{s}$ to model an external observable transition trace as $\stackrel{\hat{s}}{\Rightarrow}$.

3.3 Semimodularity

Semimodular [47] definitions are employed to define the behavior of local agents of a system in our formal verification engine in order to remove glitches in a design and enable us to locally constrain signal orderings. Figure 3.3 shows the semimodular definition of a two-input NAND gate.

A system is semimodular if and only if for all transitions, once enabled, they are not allowed to be disabled [3]. The violation of semimodular constraints on an output signal could result in a runt pulse or internal glitch in a circuit that may cause incorrect or unexpected behavior. One class of errors from the formal verification engine – computation interference – is created based on the semimodular constraints used in the specification of agents.

An extension of the original semi-modular definition is used to detect short circuit failures of dynamic gates by specifying that no transitions are valid that will result in both a p-stack and n-stack being simultaneously turned on.

This extension to support short circuit failure detection of dynamic gates is recognized by default. However, if a transient short circuit is allowed, a special state for short circuit status may be used in a formal agent definition. An example of such design is a GasP circuit that makes use of wire delays and may allow a transient short circuit.

3.4 Logic Conformance

Conformance was proposed to check if an implementation is a safe substitution of a specification in the trace theory by Dill [48]. Trace theory describes the behavior of circuit by a sequence of signal transitions which corresponds a partial history of signals. Dill developed the trace theory into a verifier and applied it a number of speed-independent asynchronous circuits [49, 50]. However, the trace conformance is too weak and cannot detect deadlock and other hazards.

Gopalakrishnan improved Dill's work and proposed *strong conformance* relation [51]. The strong conformance relation is capable of detecting deadlock. An implementation I strongly conforms to the specification S, denoted as $I \sqsubseteq S$, if implementation may offer to accept excess inputs that specification cannot accept but must be able to generate all the outputs that specification is capable of producing.

However, the trace conformance cannot distinguish nondeterminism and equate too many branching structures of agents even though strong conformance has the ability to detect deadlocks. Hence the formal verification engine used in this research employs bisimulation semantics [52, 53, 54] and is applied to the conformance relation shown in the following definition [43].

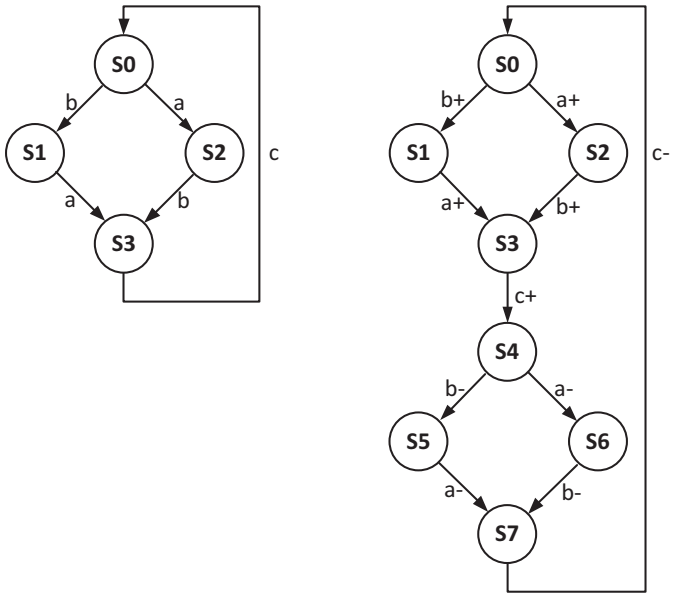
Definition 3.4 *A binary relation* $\mathcal{LC} \subseteq \mathcal{P} \times \mathcal{P}$ *over agents is a* **logic conformation** *between implementation* I *and specification* S *if* $(I, S) \in \mathcal{LC}$ *then* $\forall\ \alpha \in Act$ *and* $\forall\ \beta \in \overline{\mathcal{A}} \cup \{\tau\}$ *(outputs and* τ*) and* $\forall\ \gamma \in \mathcal{A}$ *(inputs)*

1. *Whenever* $S \xrightarrow{\alpha} S'$ *then* $\exists\ I'$ *such that* $I \stackrel{\widehat{\alpha}}{\Rightarrow} I'$ *and* $(I', S') \in \mathcal{LC}$
2. *Whenever* $I \xrightarrow{\beta} I'$ *then* $\exists\ S'$ *such that* $S \stackrel{\widehat{\beta}}{\Rightarrow} S'$ *and* $(I', S') \in \mathcal{LC}$
3. *Whenever* $I \xrightarrow{\gamma} I'$ *and* $S \stackrel{\gamma}{\Rightarrow}$ *then* $\exists\ S'$ *such that* $S \stackrel{\gamma}{\Rightarrow} S'$ *and* $(I', S') \in \mathcal{LC}$

Logic conformance is a partial order between the implementation and specification that allows multiple implementations to be conformant to the specification. They are conformant only if the implementation is a safe substitute for the implementation. Logic conformance is similar to trace conformance but still performs back and forth bisimilar relation checking between the implementation and the specification. If any of the above clauses are not satisfied, nonconformance errors are reported by the formal verification engine.

Clause 1 of Definition 3.4 specifies that if the specification can do a transition, the implementation must do the same transition. Clause 2 of Definition 3.4 says that when the implementation generates internal τ or primary output transitions, the specification must be capable of producing the same transition. Clause 3 of Definition 3.4 allows the implementation to be capable of accepting more input

transitions than the specification. In case both implementation and specification can do the same input transition, however, the specification must do the matching transition.



(a) CCS Model (b) Traditional Model

Figure 3.1. State space difference between CCS and traditional model of a C-element.

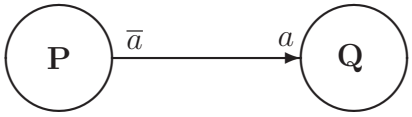


Figure 3.2. Demonstration of labels and colabels of internal transition τ.

```
*****************************************************
***           2-INPUT NAND GATE                   ***
*****************************************************
1: agent NAND001 = a.NANDa01 + b.NAND0b1;
2: agent NANDa01 = a.NAND001 + b.NANDab1;
3: agent NAND0b1 = a.NANDab1 + b.NAND001;
4: agent NANDab1 =                          'c.NANDab0;
5: agent NANDab0 = a.NAND0b0 + b.NANDa00;
6: agent NAND0b0 =             b.NAND000 + 'c.NAND0b1;
7: agent NANDa00 = a.NAND000             + 'c.NANDa01;
8: agent NAND000 = a.NANDa00 + b.NAND0b0 + 'c.NAND001;
```

Figure 3.3. Semimodular CCS specification of a 2-input NAND gate.

CHAPTER 4

AUTOMATING CONSTRAINT GENERATION

Relative timing constraints are used throughout the asynchronous circuit design flow described in the previous chapters. Asynchronous template characterization needs a key set of relative timing constraints for correct behavior of the template circuit. The set of relative timing constraints for protocol verification ensures correct interaction among multiple local asynchronous templates. The setup constraints between a control path and data path guarantee that the data are properly latched. These relative timing constraints are used for timing-driven synthesis and place and route. Finally they are used for static timing analysis with postlayout extracted parasitic parameters.

In this design flow, there are two steps that synchronous design engineers have not been involved in before – running formal verification and generating relative timing constraints. Previously generating relative timing constraints was performed manually and requires deep expertise on asynchronous circuit itself. This is absolutely an impediment for the wide adoption of this design methodology. In this chapter, an automatic method for generating relative timing constraints will be described. The designers do not need to know any details about the circuit. One push of a button can return a complete set of relative timing constraints which guarantees the correctness of the system.

This chapter first describes past related work and its weaknesses. Then the failures reported from the formal verification engine and their formal definitions are analyzed to find their common characteristics. Then a solution to generating relative timing constraints and the proper data structure are proposed. Finally the algorithms for implementing the solution are described.

4.1 Past Work

An algorithm for the automatic generating relative timing constraints was proposed in 2002 by Kim et al. [55]. The constraints generated are point-of-convergence (POC) constraints where no point-of-divergence (POD) is specified. This algorithm explores the whole state space of circuit implementation and creates state sets, called Q-sets, where failure transitions are enabled and ready to fire. Transitions which exit the state set are required to fire before the transition that produces the error, thus avoiding the timing violation. Figure 4.1 is the partial state transition graph of the GasP circuit. The symbol $\perp$ in the figure denotes failure states. The transitions directed to $\perp$ are failure transitions. In state s21 and s22, if `out`$-$ fires before `x`$-$, a failure occurs. The constraint `x`$- \prec$ `out`$-$, if applied, will remove the failure and makes the failure state unreachable. The algorithm of Kim et al. is efficient in producing POC constraints. However, it has a few weaknesses.

1. The transition set is generated from the flat transition graph of the whole implementation. This results in exponential states and loss of any hierarchical and modular information.

2. Only POC constraints are generated. The constraints specify only the relative ordering of two events. These two events may have no clue of their relationship and may not be intuitive to the designers in explaining the root cause of the failure. Without a POD, path based constraints will not be generated. As a result, the generated constraints cannot be supported for pre and postlayout timing validation using industry standard STA tools.

3. The algorithm does not support multicycle constraints or other more complicated dependencies between signal sets. Signal dependencies with logic levels specified is normally enough for a small design. As a design gets more complicated where cross cycle dependencies exists, transitions using logic levels may not be enough to represent the true intent. This dissertation proposes unrolling the behavior of an implementation to solve the multicycle problem which will be described in later chapters.

The algorithms for automatic generation of relative timing constraints that will be described in this dissertation overcome all the above weaknesses.

1. The algorithm presented in this thesis is based on signal error traces instead of the whole state graph of the implementation. It also follows the hierarchical behavior of local processes and signal sets inherited from the formal verification engine.

2. The common causal POD will be generated by backtracking the causal relationships of two relative events. The option of selecting a user-defined POD is supported as well to facilitate pre- and postlayout timing validation.

3. The signal transitions in relative timing constraints use an unrolled representation. This unrolling representation unambiguously presents cross cycle dependencies between signal events.

Another similar work was proposed by Yoneda et al. [56, 57]. This work employs metric timing and a formal verification tool VINAS-P [59], which is based on a timed version of trace theory verification [58] by using partial order reduction based on Dill's work [48]. Timed Petri nets are used to model both specification and circuit implementation. Initially the min-max bounds of delay of a gate is set to be sufficiently large and safety properties are checked within VINAS-P tool for any hazard, hold time violation and short circuits. If it detects a failure, an error trace is returned. Based on the error trace, some form of delay bound relationship is derived to avoid the failure. ILP (Integer Linear Programming) solver is used to generate tightened min-max bounds of gate from the delay bound relationship. Then the min-max delay is updated with tightened one and run formal verification recursively until no errors are reported.

The process of generating delay bound relationship implicitly specifies a relative timing constraints by finding two events and enforcing firing order of them to avoid failure. It also backtracks the common causal transition of those two events. Thus the firing order of two events forms the delay bound relationship. This relationship is used to generate tightened bound by ILP. However, the tightened delay bounds of gates may over constrain the gates along the path from their common causal point

to failure events. Relative timing, on the other hand, constrains the whole path from point of divergence to point of convergence. The specific delays of gates and wires along the path are left, manipulated and optimized by synthesis and place and route engines without any concern about any particular delay bound on a gate.

4.2 Formal Definitions

The functionality of a relative timing constraint is to enforce precedence over two events such that the failure state directed by the preceded event is not reachable. Hence generation of relative timing constraints is directly related to how the failure is generated and where the failure states are located.

The formal verification engine *Analyze* reports three major classes of errors – computation interference, nonconformance and deadlock. The following sections formally define each class of errors. This helps better understanding of the underlying failure mechanisms.

4.2.1 Computation Interference

Computation interference is implemented based on the CCS parallel composition operator which is denoted as "|". A design is generally modeled as a set of processes connected with parallel composition operators. A process represents the state of an agent which is either a primitive gate or protocol. The composition of processes guarantees that parallel agents can evolve concurrently.

Communication among connected parallel agents within the design is an autonomous transition denoted as the internal transition τ. When the output of a predecessor element fires, denoted as *colabel* $\overline{\alpha} \in \mathbf{sort}(P_{pred})$ the corresponding *labels*, denoted as $\beta \in \mathbf{sort}(P_{succ})$, which are inputs of the receiving elements connected with the *colabel* evolve simultaneously. A successful internal transition between parallel agents is formally described as $\overline{\alpha} = \beta \wedge (P_0 \mid P_1 \mid \ldots \mid P_n) \xrightarrow{\tau} (P_0' \mid P_1' \mid \ldots \mid P_n')$ where

- for agent whose colabel is $\overline{\alpha}$, $P_i \xrightarrow{\overline{\alpha}} P_i'$
- $\forall P_j : \ \beta \in \mathbf{sort}(P_j)$, P_j must be in a state where $P_j \xrightarrow{\beta} P_j'$
- for all other processes $P_k = P_k'$

Computation interference violates the above formula. It results from an unacceptable signal transition or *label* to a process. Agents are modeled as either terminal semimodular specifications or as a minimized specification of a protocol. Figure 3.3 shows a semimodular specification of a 2-input NAND gate with inputs `a` and `b` and output `c`. For those agent states where an input transition is not specified, they are unacceptable in the current state. When some element outputs and initiates an internal transition, if the receiving element does not specify the corresponding input transition at current state, a computation interference error is reported. Agent NANDab1 on line 4 of Figure 3.3 does not specify transitions for input signals `a` and `b`, which means that computation interference occurs whenever there is such an input transition at state NANDab1. At state NANDab1 the only transition the agent can make is firing output `c`.

Perhaps the state transition graph of the two-input NAND gate specification shown in Figure 4.2 is more intuitive. The signal transitions directed to the horizontal bars are unacceptable transitions. The horizontal bars represent failure states which should be avoided.

The **dynamic** set defines all the enabled and ready-to-fire signal transitions of the circuit in a given state (I', S') after a trace is executed. For each signal transition in the trace, there is a corresponding **dynamic** set specifying the signal transitions the circuit can make. The signal transitions in the trace and their corresponding **dynamic** set forms a partial directed acyclic state transition graph. It is called partial because all the other branches other than the trace signal transitions are omitted. Figure 4.3 shows such a graph where $s = abc$. This graph is actually a flattened state transition graph of the top level system. Starting from state S0, the trace $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{c}$ is executed to reach a failure state. At state S0, the **dynamic** set consists of $\mathtt{a}, \mathtt{b}, \mathtt{c}$. At state S3, the **dynamic** set consists of $\mathtt{c}, \mathtt{d}$. It is different from the state transition graph of the local element shown in Figure 4.2. The **dynamic** set is formally defined as follows.

Definition 4.1 **dynamic**(I', S') *is the action sequence of inputs and outputs* $\overline{\alpha} \cup \beta$ *possible from the current implementation state* $I' = (P'_0 \mid P'_1 \mid \ldots \mid P'_n)$ *after a trace* s *is executed where* $I \xrightarrow{s} I'$*, and if* S *exists,* $S \stackrel{\widehat{s}}{\Rightarrow} S'$ *such that*

- $\forall P_i' \xrightarrow{\overline{\alpha}} \ \wedge \ \overline{\alpha} \in \overline{\mathcal{A}} \cup \tau$
- *If* $\exists S', \ \forall S' \stackrel{\widehat{\beta}}{\Rightarrow} \ \wedge \ \beta \in \mathcal{A}$

Computation interference is formally defined as follows.

Definition 4.2 *Computation Interference occurs when an agent P_i in implementation $I' = (P_0' \mid P_1' \mid \ldots \mid P_n')$ and its associated specification S', if it exists, cannot accept input α:*

- $\alpha \in \mathbf{dynamic}(I', S') \wedge \alpha \in \mathbf{sort}(P_i') \wedge P_i' \stackrel{\alpha}{\not\rightarrow}$

The relative timing constraints can be generated from either a local or global state transition graph. Intuitively to avoid reaching the failure state, some enabled event should occur before the event which causes the failure. This enforced ordering is allowed in that *Analyze* performs untimed verification which assumes unbounded gate delay for speed-independent verification plus unbounded wire delay for delay-insensitive verification. Thus the circuit model contains huge concurrency. Two concurrent signal transitions can fire in either order [61]. Relative timing is the constraint that forces the firing order of two events such that the reachability of the failure state is avoided. In Figure 4.3 the second transition of `c` causes a failure. Always firing the first transition of `d` before the second transition of `c` guarantees the failure state is never reachable.

There are two main sources of computation interference in a logic gate.

1. An input transition is trying to disable an output transition.
2. A short circuit failure in a dynamic gate by turning on both pull-up and pull-down networks at the same time.

4.2.2 Nonconformance

Nonconformance is applied only when the conformance between circuit implementation and specification is checked. The verification engine employs the bisimulation conformance relation shown in Definition 3.4 for observable input and output signals.

Nonconformance has two subtypes of errors. One is reporting an illegal output. This is to say that the circuit implementation can generate an output but the specification is not able to do this output transition. It violates the second statement in Definition 3.4 (Whenever $I \xrightarrow{\beta} I'$ then $\exists\ S'$ such that $S \overset{\widehat{\beta}}{\Rightarrow} S'$ and $(I', S') \in \mathcal{LC}$).

Definition 4.3 *An illegal output occurs at (I', S') where $I \xrightarrow{s} I'$ and $S \overset{\widehat{s}}{\Rightarrow} S'$ after trace s is executed if $\exists \alpha \in$ **dynamic**$(I', S')\ \wedge\ \alpha \in \overline{\mathcal{A}}$, such that $I' \xrightarrow{\alpha} I''$ and $S' \overset{\alpha}{\not\Rightarrow}$.*

When the circuit implementation generates an output that the specification cannot perform, there are three possibilities from both implementation and specification as follows.

- The current state of the specification allows only an input transition to occur. This input transition is in the **dynamic** set.
- Another output is desired by the current state of the specification. Since the output is generated from a logic gate, there must be some controlling signal in the **dynamic** set that is causal to the desired output.
- Some internal transition, if it fires, can disable the illegal output from occurring at all.

All the above three possibilities can be solved by firing **dynamic** set signals before any hazard transitions.

The second subtype of nonconformance error is that a primary input/output transition is required by the specification but is not possibly generated by the implementation.

Definition 4.4 *A primary input/output transition α is required by the specification but it is not possible for implementation to generate the same transition at (I', S') where $S' \overset{\widehat{\alpha}}{\Rightarrow} \wedge\ I' \overset{\alpha}{\not\rightarrow}$.*

This type of error belongs to the category of the unsolvable set of problems. One possible conclusion is that the implementation cannot be made conformant to the specification by adding timing constraints. Another conclusion is that if such an

error occurs when applying relative timing constraints, relative timing constraints could be the cause of error if they are so strong that states which should be reachable are made unreachable. If such error occurs without any interference of relative timing constraints, it must be a defective design.

4.2.3 Deadlock

Deadlock occurs when two or more processes in a loop wait for each other's triggers to proceed. The system gets stuck at a particular state and no legal transition available can make the design proceed. Deadlock can be checked by tool *Murphi* [60]. Deadlock in our formal verification engine indicates that all the enabled and ready-to-fire signals are blocked by their receiving agents, thus no available transitions can make the system proceed.

Definition 4.5 *Deadlock occurs at* (I', S') *where* $I \xrightarrow{s} I'$ *and* $S \stackrel{\hat{s}}{\Rightarrow} S'$ *after trace* s *is executed iff* $\forall \alpha \in \mathbf{dynamic}(I', S')$

- *if* $\alpha \in \overline{\mathcal{A}}$ *(output), then* $S' \stackrel{\alpha}{\not\Rightarrow}$
- *if* $\alpha \in \mathcal{A}$ *(input)* $\ \wedge\ \alpha \in \mathbf{sort}(P'_i)$*, then* $P'_i \stackrel{\alpha}{\not\rightarrow}$
- *if* $\alpha \in \tau\ \wedge\ \tau(\overline{\alpha}) \in \mathbf{sort}(P'_j) :\ \tau(\alpha) \in \mathbf{sort}(P'_k)\ \wedge\ P'_j \xrightarrow{\overline{\alpha}} P''_j\ \wedge\ P'_k \stackrel{\alpha}{\not\rightarrow}$

At a particular state (I', S') where $I' = (P'_1 \mid P'_2 \ldots \mid P'_n)$, if all the enabled signal transitions in the **dynamic** set cannot fire due to being blocked by its subsequence agents, a deadlock error will be reported. In the **dynamic** set, if one action is a primary output signal, it must be blocked by the current state of the specification; if it is a primary input signal or an invisible internal signal, it must be blocked by its receiving agents. Hence no other signal can make the system proceed, i.e. $(I', S') \stackrel{\alpha}{\not\rightarrow}$ and the system is in an awkward interlock state. Figure 4.4 shows an illustration of deadlock. The three state graph in square is a simplified state graph to represent the specification. Below the dashed line, is a simplified partial implementation that contains only related agents. Transition α is the primary output blocked by specification. Transition β is the primary input blocked by P'_i. Transition τ is the internal transition blocked by P'_j.

The deadlock error appears together with computation interference and illegal output errors. If the enabled signal in the **dynamic** set is a primary output, its firing creates an illegal output because the circuit implementation generates an output that the specification cannot perform. On the other hand, if the enabled signal in the **dynamic** set is a primary input or an internal transition, the receiving agent's blocking behavior is actually a computation interference.

Therefore the handling of deadlock errors is summarized as follows:

1. If deadlock errors appear with computation interference or nonconformance errors, solve computation interference or nonconformance errors.

2. If deadlock errors appear without computation interference or nonconformance errors when some relative timing constraints have already been enforced on the design, the problem is unsolvable because the interference of relative timing constraints may over constrain the design and causes errors. This set of relative timing constraints should be discarded.

3. If deadlock errors appear without computation interference or nonconformance errors and there is no interference of relative timing constraints, it is a defective design.

4.3 Common Feature of Hazards

The solvable set of hazards consists of computation interference and illegal output of nonconformance errors. If neither of them appears in the error report, the design is either defective or over-constrained by relative timing constraints.

Computation interference occurs when tokens hit an unacceptable failure state of a local element of the implementation. The illegal output is generated by the implementation but forbidden by the specification, thus the egress arc of illegal output transition points to a virtual failure state. Both computation interference and illegal output errors can be mapped by a template state transition graph of the implementation shown in Figure 4.5.

- The horizontal bar indicates a failure state that is reached from transition α_{fail} at the current state I'.

- Transition α_{fail} is the failure transition. In the case of computation interference, α_{fail} is the signal transition that is not acceptable by the process. In the case of an illegal output, α_{fail} is the illegal output itself.
- I' is the implementation process where the failure transition becomes enabled.
- Signal α_{en} is the transition that moves process from I to I' where $I = (P_1 \mid P_2 \mid \ldots \mid P_n)$ and $I' = (P_1' \mid P_2' \mid \ldots \mid P_n')$ since $I \stackrel{\alpha_{en}}{\rightarrow} I'$.
- **dynamic**$(I, S) = \cup_{i=1\ldots m}\alpha_{m-1} \cup \alpha_{en}$.
- **dynamic**$(I', S') = \cup_{i=1\ldots n}\beta_{n-1} \cup \alpha_{fail}$.

The fundamental idea is to avoid reaching the failure state from the perspective of this partial transition graph to prevent the failure transition from occurring by firing another concurrently enabled transition. This is exactly what the relative timing constraint intends to do. The **dynamic** set contains all the enabled signal transitions at a particular state of the implementation. Any signal transition in the **dynamic** set can be forced to occur prior to the failure transition. Hence a single error from the formal verification engine may return one or more candidate relative timing constraints.

The path based relative timing constraint is represented as POD/POC pair. The point-of-divergence (POD) indicates a starting point that initiates the race condition. The point-of-convergence (POC), which is composed of a relative ordering normally converges to a single gate to indicate where and how the race condition occurs. However, the relative ordering of two race events is not necessarily converged to a single gate as long as it is supported by postlayout timing validation. Strict POD/POC constraints are more friendly to the designer because the designer can easily figure out the location and cause of the racing condition. Strict POC constraints can be achieved by constraining the signal transitions in the **dynamic** set to be in the **sort** set of the element where the failure occurs, i.e., $\alpha \in$ **dynamic**$(I', S') \wedge \alpha \in$ **sort**(P_i).

Figure 4.5 also adds a second level state that is directed by transition α_{en}. The **dynamic** set at state I can also fire before transition α_{en}. This set of constraints is stronger than the one derived at state I' since it makes state I' unreachable.

Stronger relative timing constraints remove more subgraphs and can result in a more compact set of constraints to make the implementation conform to the specification. A stronger constraint, however, may over-constrain a design and cause unexpected errors. Weaker constraints, on the other hand, always removes the state graph closest to the failure state and guarantees the correctness of the design under RT constraints, but the cardinality of the final set of relative timing constraints may be bigger, which may increase the burden of pre- and postlayout timing validation. Choosing an optimal set of relative timing constraints is future work.

4.4 Generating Relative Timing Constraints

Path based relative timing constraints are composed of the relative ordering of two racing events and their common causal point POD. Relative ordering can be created by finding the key signal transition that has to be preceded and its corresponding **dynamic** set. The POD can be found by backtracking based on the causal relationship of race events.

Figure 4.6 is the top level algorithm for automatic relative timing constraint generation. TST stands for Trace Status Tableau, which is the internal data structure that contains all necessary information inherited from the formal verification engine *Analyze*. The trace status tableau is built along the trace. Once it is constructed, all the key information corresponding to Figure 4.5 such as α_{fail}, α_{en}, I, I', and the **dynamic** sets for I and I' can be found to construct relative orderings. The structure of trace status tableau will be described in the next section. The point-of-divergence can be backtracked by tracing causalities in the trace status tableau. A user-defined POD is supported during POD backtracking to facilitate postlayout timing validation. The controlling signals in the **dynamic** set determine the destination container that stores the relative timing constraints. This depends on the value of the input parameter POCConstrOption of the algorithm. If it is true, the algorithm only returns the relative timing constraints whose relative ordering converges to a single gate. Otherwise, the algorithm returns all possible candidate relative timing constraints.

4.5 Trace Status Tableau

A trace status tableau contains all necessary information for generating relative timing constraints. Each verification error corresponds to one tableau. A trace status tableau is constructed based on the counter example trace returned from the formal verification engine. Along the trace, each wire node updates its status. Wire nodes can be regarded as state holding elements. The tableau can be regarded as a two-dimensional array with signal transitions of the trace as x-axis index, and all wire nodes as y-axis index.

Each cell in tableau contains the necessary information that associates with the x-axis and y-axis indices. It contains the current status of the wire nodes including signal state, number of transitions this signal has already made, whether this signal has been enabled and is ready to fire, and whether a failure occurs. Table 4.1 shows a simplified example trace status tableau of the trace $(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{a})$ for illustration purpose only.

The y-axis lists all wire node signals in the design trace. The wire node signals can be classified into three categories.

- Primary input signals. All primary input signals shares a single state transition graph derived from the specification.

- Primary output signals. The status of an output signal can be tracked by the action of the agent that drives the output. Note that when an output fires, the tokens of the state transition graph of the specification changes as well. This may affect the state of primary inputs in case of conformance verification between a specification and implementation.

- Internal signals. An internal signal connects the output of one element to the input of another or the same element. The element whose output is the internal wire is called predecessor whereas the elements the internal wire feeds into are called successors. For self-loop where the output of an element feeds back into its own input, the predecessor and successor are the same element. When an internal signal changes, both predecessor and successor update their states.

In a trace status tableau, each cell associates the x-axis signal transition of the trace and the y-axis wire node signal. The information contained in the cell which is shown below records the current status of wire node signal.

- **State**: current state of corresponding wire node signal.
- **Number of transitions**: the number of transitions the wire node signal has already made.
- **Enabled flag**: a bool indicating whether the wire node signal is enabled.
- **Failed flag**: a bool indicating whether a failure occurs at the current element associated with the corresponding wire node signal.

Figure 4.7 shows the algorithm for generating the information of a cell of the trace status tableau. Generating a cell relies mostly on the status of its previous cell. First the previous cell information is inherited. Then each type of information of the new cell is generated by its specific functions. The details of these functions are described in the following subsections.

For the same design, the size of trace status tableau depends on the length of error trace. *Analyze* uses a breadth-first algorithm to find errors. Thus it always returns the shortest error trace. As a result, there is no concern about the trace status tableau size.

4.5.1 State

Each wire node signal is associated with a semimodular state transition graph for generating its state. All the primary input signals share the state transition graph of the specification. The tokens move whenever an input or output signal fires. The states of the primary outputs or internal signals are based on the status of the logic elements they associate with. The state transition graph of each element is mapped from its semimodular definition modeled in CCS. Figure 3.3 is the semimodular definition of a two-input NAND gate and Figure 4.2 is its state graph representation with explicit failure states specified.

The state of a wire node is updated along the trace. If the wire node signal is not related to the current trace signal transition, it remains its previous state. Figure 4.8

describes how to find the next state of a wire node signal based on its previous state and signal transition of trace. TraceSig denotes the current signal transition of the trace. WireSig denotes the wire node signal. PrevState is previous state of WireSig when executing previous TraceSig. Generating next states is performed based on the categories TraceSig belongs to. If TraceSig is an internal signal and also belongs to the **sort** of WireSig, the next state of WireSig can be found by calculating the behavior of corresponding WireSig with its PrevState and the transition TraceSig. If the trace signal transition is a primary input, the local elements who are connected to this input transition update their states. In the meanwhile, it updates all the primary input wire node signals since they share the common behavior of the specification. Finally if the trace signal is a primary output signal, all the primary input and output wire node signals update their states. The primary output wire signal is updated based on the behavior of the local element that is its source. The primary input wire signals are updated based on the behavior of the specification.

4.5.2 Number of Transitions

The number of transitions records how many transitions the wire signal has made along the trace. This number is used to generate the unrolling count of signal transitions to support multicycle constraints. One of the advantages of using unrolling counts compared to logic levels is that unrolling counts which are specified after the signal name clearly indicates the history of signal transitions. Given a logic level, on the other hand, one cannot guarantee if this transition occurs the first time or multiple times. For example, suppose the initial logic level of signal `a` is low. The unrolling representation `a` 0 and `a` 2 indicate the first and the third transition of signal `a` but `a` 0 and `a` 2 both represent the same logic level `a`+. Therefore the unrolling representation of signal transition has implicit timing relations in it.

The design methodology described in this thesis is not restricted to asynchronous design but supports clocked design. The unrolling representation of signal transitions is more powerful when verifying a clocked design where cycle accurate transitions are required. Figure 4.9 illustrates the usage of the unrolling representation for a signal transition in a clocked design. If one wants to specify a timing constraint

as `data`+ $\prec$ `clk`+, it might be a misleading representation because `data`+ $\prec$ `clk`+ indicates that `data` is high before *every* positive edge of `clk`. Hence the timing of the example in Figure 4.9 cannot be accurately represented by such logic level format. The unrolling representation, as an alternative, uses `data` 0 $\prec$ `clk` 2 to represent the timing assumption in Figure 4.9. It is to say that signal `data` first goes high before the second transition of signal `clk`. It is exactly the second transition of `clk` but not the fourth or the sixth etc..

Figure 4.10 shows the algorithm for generating transition counts in the trace status tableau. The transition count increments only when the trace signal matches the wire node signal.

4.5.3 Enabling and Causal Relations

The so called "enabled" bit is defined as *enabled* but not yet fired. The primary outputs and invisible internal transitions τ that are output ports of local gates are *enabled* by a particular pattern of its input logic value. The primary inputs are *enabled* as the environment requires.

Figure 4.11 shows the algorithm for generating the *Enabled* flag. Every signal transition in the trace resets the *enabled* flag of its corresponding wire node signal in the y-axis to be false because the occurrence of the signal in the trace means that the signal has fired. If the trace signal is the input port of a local gate whose output port is a wire node signal and its current input port value is in such a pattern that its output is *enabled*, the *Enable* should be flagged to be true. This is implemented by searching the wire node signal in the action set of the current state. The action set consists of enabled signals of a single process which is inherited from *Analyze*. Since the *Enabled* flag is generated based on its current state instead of previous state, it has to be performed after the current state is generated.

The *Enabled* flag is the key information to finding causal relationship of events. If a cell in the tableau has its *Enabled* flag true and its previous cell has its *Enabled* flag false, the wire signal is said to be enabled by the current cell associated with the x-axis trace signal transition. This helps find the point-of-divergence which will be described in the section of POD backtracking.

4.5.4 Locating Failure

The *Failed* flag in a cell, if true, indicates that a failure occurs at the local gate whose output is its corresponding wire node signal. This local gate is the point-of-convergence gate. If strict POC constraints are required, the **sort** of the POC agent is used to filter candidate relative timing constraints.

The formal verification engine returns an error trace where the last signal transition in the trace is always the transition that causes the failure. It could be an unacceptable transition to a gate or an illegal output against the specification. Therefore only the last status cells of wire node signals may have the *Failed* bit flagged. The generation of the *Failed* flag does not depend on its previous cell information.

Figure 4.12 shows the algorithm for generating the *Failed* flag of a cell. When *Failed* is flagged is directly related to the definitions of computation interference and illegal output errors. For a computation interference error, the trace signal transition is not accepted by a process and thus is not contained in the action set where legal enabled transitions are specified. The illegal output itself is the failure transition, thus the status cell of its corresponding wire node signal should set the *Failed* flag to be true.

4.6 Relative Ordering

The relative ordering specifies a safety property that one of the events *always* occurs before the other. It is a key part of path based relative timing constraints. The generation of relative ordering described in this thesis is similar to hand generation – firing of another enabled transition before the failure transition, thus failure state is never reachable.

Note that the relative ordering is not disabling the occurrence of failure transition themselves but disables the ability to reach failure state. The failure transition, once enabled, has to fire and fires after the controlling signal transition if this constraint is applied to the design. Figure 4.13 clearly shows an example of relative ordering $\texttt{b}+ \prec \texttt{a}-$. Transition $\texttt{a}-$ leads to the failure state. Firing $\texttt{b}+$ and then $\texttt{a}-$ prevents

reaching a failure state and leads to the good state S4. The failure transition a− at state S1 is not disabled.

Relative orderings can be compared in terms of their relative strength. The relative strength of relative orderings is defined based on the flattened state transition graph of the system where the root node is the initial state of the implementation and the children nodes are consequent states directed by legal transitions as arcs. For relative ordering A to be stronger than relative ordering B it must satisfy the following conditions.

- Relative ordering B, if applied to the state transition graph, removes a failure.
- Relative ordering A, if applied to the state transition graph, is able to remove the same failure.
- The subgraph that A removes contains the branch at which B is applied.

Figure 4.14 shows an example of relative ordering strength. Relative ordering a− $\prec$ b+ is stronger than c− $\prec$ d+ because a− $\prec$ b+ removes the subgraph that already covers the relative ordering c− $\prec$ d+ and makes c− $\prec$ d+ be a redundant constraint.

Stronger constraints can cover weaker constraints and will result in a more compact set of final relative timing constraints. Compared to the weaker constraint, a stronger constraint cuts the state graph at a higher level which is closer to the root node, and thus may over-constrain the design and cause unexpected errors. Weaker constraints, on the other hand, guarantee the correctness while not over-constraining the design, but results in more constraints. The number of relative timing constraints directly determines the burden of pre- and postlayout timing validation.

The algorithms for generating relative timing constraints returns weaker constraints. Relative timing constraints are always generated near the failure point. In the real scenario the behavior of a component is restricted by its adjacent environment. Only a partial behavior will be passed through and others are not possible upon the restriction of environment. Although weaker constraints resolve the errors, a large amount of states are still not reachable. A better relative timing constraint could be a constraint that not only removes the error but also happens to remove unreachable states without over-constraining the design. This problem will be demonstrated in

the example section. Choosing the best relative timing constraint is left for future work.

According to the top level algorithm in Figure 4.6, a bunch of information such as α_{fail}, α_{en}, current state, previous state, and the **dynamic** set at the current and previous states needs to be generated to construct relative ordering.

The failure transition α_{fail} is normally the last signal transition in the error trace because the verification engine halts when an error occurs. Since CCS does not distinguish the logic levels of transitions, only a list of signal names in sequence are returned from the formal verification engine. Hence the trace is processed and a transition count is added to each corresponding signal transition of the trace.

As the trace status tableau is a two-dimensional array, indexing is used to locate a cell for finding any necessary information. Figures 4.15, 4.16, 4.17, 4.18 and 4.19 show how to generate the failure transition, current state, previous state, enabling transition and the **dynamic** set respectively. The failure transition is just the last element of the trace. The current state of the POC can be located as the cell where the *Failed* flag is true. The previous state of the POC can be traversed backward horizontally to find any change of state. The enabling transition that changes the POC state from the previous to the current one can simply be derived from the x-index of previous state. By traversing all the wire node signals at the index of state, the enabled signals are all added to the **dynamic** set.

The transition α_{fail} is associated with **dynamic**(curState) while transition α_{en} is associated with **dynamic**(prevState). The relative constraints constructed at α_{en} are stronger than the ones at α_{fail}. The constraints at α_{en} must be considered in case that there is no controlling signals at α_{fail}, i.e. **dynamic**(curState) is an empty set.

4.7 POD Backtracking

Pre- and postlayout timing validation must be performed to validate that the constrained timing holds with extracted parasitic parameters. Static timing analysis using commercial CAD tool such as Primetime employs clocked algorithms which requires a clock signal as a global reference.

To validate the relative timing constraints of asynchronous circuits using command set_data_check, a virtual clock must be specified with the -clock option. The POD of path based relative timing constraints is used as the virtual clock. Generally this virtual clock is mapped to the input port of a module, e.g., the request signal of handshake controller.

The algorithms described in this thesis for the automatic generation of relative timing constraints supports user defined point-of-divergence. By default, the algorithms returns the first or the closest matching causal signal as point-of-divergence.

Figure 4.20 presents the algorithm for finding the point-of-divergence. It takes the x-axis and y-axis indices of the two transitions in relative ordering, trace status tableau and the user defined POD and returns a desired POD. The full causal list of each event is generated by the GenCausal subroutine and then the MatchPOD subroutine finds the point-of-divergence either by default with nothing specified for UserDefPOD or by matching against the user specified POD.

Normally all the traces of a reactive system start with its first receiving transition from the environment. For an asynchronous handshake controller, a request is always the first transition. Hence any signal transition in the trace can be backtracked down to the first transition as a causal relationship.

Backtracking to find the POD employs logical causal relationships which is different from the concept of enabling transitions described for relative ordering generation. The enabling transition moves from one agent to another. It is incorporated as the state change of an agent but does not necessarily enable an output to fire. Logical causal relationships, on the other hand, refers to when a signal enters an unstable state by some transition and becomes ready to fire. Figure 4.21 describes the algorithm for generating a full list of causal signals given a transition. It takes the index of a signal transition that one wants to backtrack and the trace status tableau and returns an array of signal transitions that sequentially lists all causal events from the end to the very beginning. This is a process that recursively traces causal signal transitions backward. The causal signal transition just found is fed back into the algorithm itself to find its parent causal transition recursively until it hits the beginning of the trace.

The full causal path of transitions is reversely stored where the original relative ordering event is stored in the last position of array, its direct causal event is stored in the second to the last position and the very first causal one is stored in the first position. The length of the full causal path may be different for the two events of the relative ordering. The reverse storage guarantees the perfect alignment of causal events. Figure 4.22 shows the algorithm for matching POD events from two causal lists of transitions. First all the common causal transitions from the beginning are recorded in an array. It is normal to have a couple of common causal points of divergence for two racing events. If the user does not define his/her desired POD, the closest POD is returned. Otherwise it either returns the user defined POD or reports that an error and exits if no matched user defined POD is found.

Each computation interference or nonconformant illegal output error may have more than one candidate solution relative timing constraint depending on concurrency at the failure point. Since the formal verification engine performs untimed verification, all the enabled and ready-to-fire signals can fire in arbitrary orders. The candidate relative timing constraints for removing a single error are mutual exclusive. Thus only one of them is fed back into the formal verification engine. While all the possibilities need to be evaluated, the solution set of relative timing constraints grows like a tree.

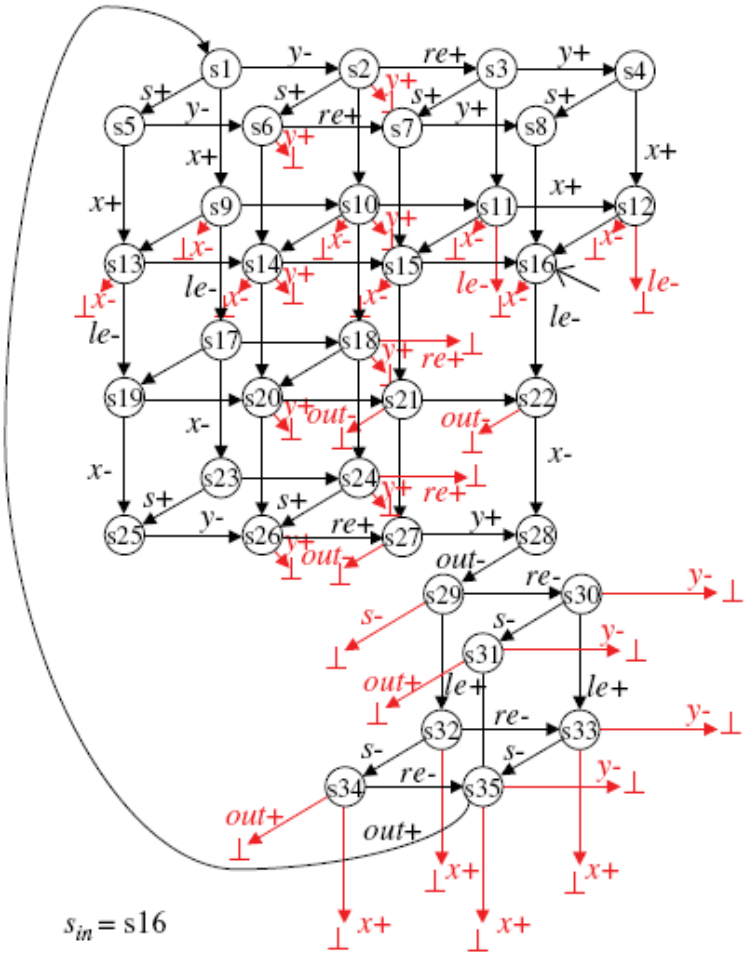


Figure 4.1. Partial state graph of GasP circuit.

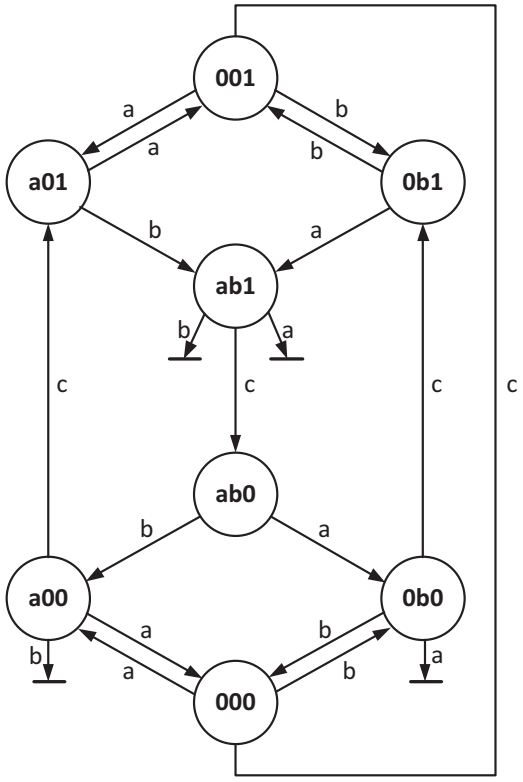


Figure 4.2. Semi-modular state transition graph of 2-input NAND gate.

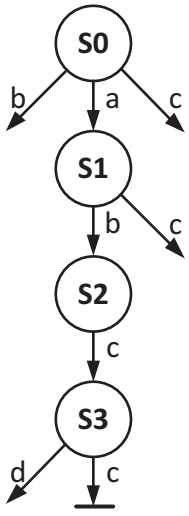


Figure 4.3. An example of flattened STG.

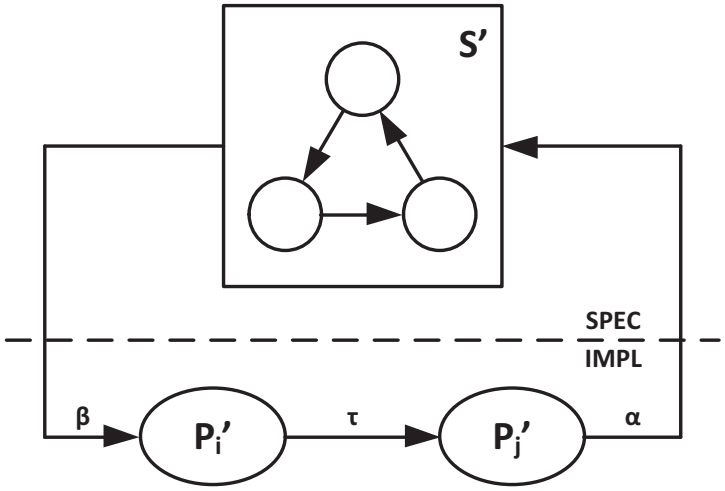


Figure 4.4. An illustration for deadlock.

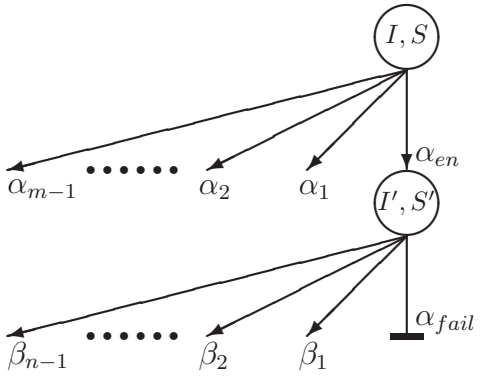


Figure 4.5. Template graph for mapping failure points.

Procedure RTGen (POCConstrOption, UserDefPOD, *Analyze*);

```
1: TST ← GenTST(Analyze);
2: α_fail ← GenFailTrans(Analyze::Trace);
3: curState ← GenCurState(TST, α_fail);
4: prevState ← GenPrevState(TST, curState);
5: α_en ← GenEnTrans(prevState);
6: curDynamicSet ← GenDynamic(curState, TST, Analyze::WireSigSet);
7: prevDynamicSet ← GenDynamic(prevState, TST, Analyze::WireSigSet);
8: for all α ∈ curDynamicSet ∧ α ≠ α_fail do
9:    pod ← GenPOD(α, α_fail, TST, UserDefPOD);
10:   if α ∈ sort(curState) then
11:      push (POCRT, pod ↦ α ≺ α_fail);
12:   else
13:      push (nonPOCRT, pod ↦ α ≺ α_fail);
14:   end if
15: end for
16: for all α ∈ prevDynamicSet ∧ α ≠ α_en do
17:   pod ← GenPOD(α, α_en, TST, UserDefPOD);
18:   if α ∈ sort(prevState) then
19:      push (POCRT, pod ↦ α ≺ α_en);
20:   else
21:      push (nonPOCRT, pod ↦ α ≺ α_en);
22:   end if
23: end for
24: return (POCConstrOption) ? POCRT : (POCRT ∪ nonPOCRT);
```

Figure 4.6. Top level algorithm of ARTIST.

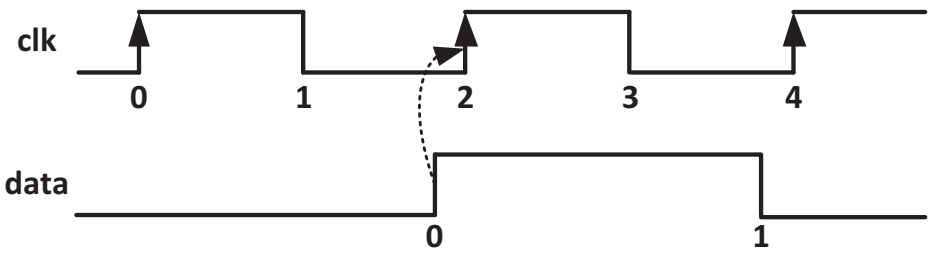
Procedure GenCell (WireSig, TraceSig, PrevCell, ErrType);

```
1: PrevState ← PrevCell.state;
2: PrevNumOfTrans ← PrevCell.numOfTrans;
3: PrevEnabled ← PrevCell.enabled;
4:
5: NxtState ← GenNxtState (WireSig, TraceSig, PrevState);
6: NxtNumOfTrans ← GenNxtNumOfTrans (WireSig, TraceSig,
                                     PrevNumOfTrans);
7: NxtEnabled ← GenNxtEnabled (WireSig, TraceSig, PrevEnabled, NxtState);
8: Failed ← GenFailed (WireSig, TraceSig, NxtState, ErrType);
9: return {NxtState, NxtNumOfTrans, NxtEnabled, Failed};
```

Figure 4.7. Algorithm for constructing the cell of trace status tableau.

```
Procedure GenNxtState (WireSig, TraceSig, PrevState);
 1: NxtState ← PrevState;
 2: if TraceSig ∈ InternalSigSet then
 3:    if TraceSig ∈ sort(PrevState) then
 4:       NxtState ← WireSigSTG(PrevState, TraceSig);
 5:    end if
 6: else if TraceSig ∈ PrimaryInputSet then
 7:    if WireSig ∈ PrimaryInputSet then
 8:       NxtState ← SpecSTG(PrevState, TraceSig);
 9:    else if TraceSig ∈ sort(PrevState) then
10:       NxtState ← WireSigSTG(PrevState, TraceSig);
11:    end if
12: else if TraceSig ∈ PrimaryOutputSet then
13:    if WireSig ∈ PrimaryInputSet then
14:       NxtState ← SpecSTG(PrevState, TraceSig);
15:    else if TraceSig ∈ sort(PrevState) then
16:       NxtState ← WireSigSTG(PrevState, TraceSig);
17:    end if
18: end if
19: return  NxtState;
```

Figure 4.8. Algorithm for generating next state.



data 0 ≺ clk 2

Figure 4.9. Timing graph of unrolling representation of signal transition for clocked system.

```
Procedure GenNxtNumOfTrans (WireSig, TraceSig,
                                        PrevNumOfTrans);
 1: NxtNumOfTrans ← PrevNumOfTrans;
 2: if WireSig eq TraceSig then
 3:    NxtNumOfTrans ← PrevNumOfTrans + 1;
 4: end if
 5: return  NxtNumOfTrans;
```

Figure 4.10. Algorithm for generating transition count.

```
Procedure GenNxtEnabled (WireSig, TraceSig, PrevEnabled,
                                        CurState);
 1: NxtEnabled ← PrevEnabled;
 2: if PrevEnabled then
 3:    if WireSig eq TraceSig then
 4:       NxtEnabled ← false;
 5:    end if
 6: else
 7:    if TraceSig ∈ sort(CurState) ∧ WireSig ∈ ActionSet(CurState) then
 8:       NxtEnabled ← true;
 9:    end if
10: end if
11: return  NxtEnabled;
```

Figure 4.11. Algorithm for generating Enabled bit.

```
Procedure GenFailed (WireSig, TraceSig, CurState, ErrType);
 1: Failed ← false;
 2: if ErrType eq COMPUTATION_INTERFERENCE then
 3:    if TraceSig ∈ sort(CurState) ∧ TraceSig ∉ ActionSet(CurState) then
 4:       Failed ← true;
 5:    end if
 6: else if ErrType eq ILLEGAL_OUTPUT then
 7:    if WireSig eq TraceSig then
 8:       Failed ← true;
 9:    end if
10: end if
11: return  Failed;
```

Figure 4.12. Algorithm for generating Failed bit.

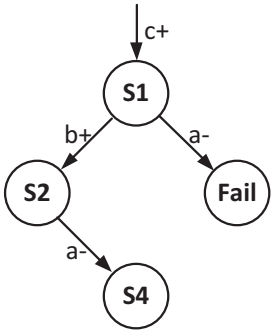


Figure 4.13. A demonstration of failure transition.

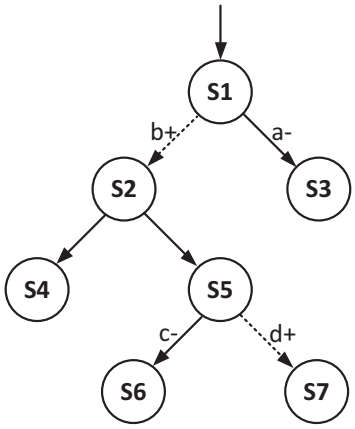


Figure 4.14. An example to illustrate the strength of relative orderings.

```
Procedure GenFailTransIndex (Trace);
  return  sizeof (Trace) - 1;
```

Figure 4.15. Algorithm for generating failure transition.

```
Procedure GenCurState (TST, FailTransIndex);
1: curStateIndex.x ← FailTransIndex;
2: curStateIndex.y ← 0;
3: if TST[FailTransIndex][yIndex].FailedFlag == true then
4:    curStateIndex.y ← yIndex
5: end if
6: return  curStateIndex;
```

Figure 4.16. Algorithm for generating current state.

```
Procedure GenPrevState (TST, curStateIndex);
1: prevStateIndex.x ← curStateIndex.x;
2: prevStateIndex.y ← curStateIndex.y;
3: if TST[xIndex][curStateIndex.y].state !=
     TST[curStateIndex.x][curStateIndex.y].state then
4:   prevStateIndex.x ← xIndex;
5: end if
6: return prevStateIndex;
```

Figure 4.17. Algorithm for generating previous state.

```
Procedure GenEnTrans (prevStateIndex);
  return prevStateIndex.x + 1;
```

Figure 4.18. Algorithm for generating enabling transition.

```
Procedure GenDynamic (stateIndex, TST, WireSigSet);
1: for yIndex = 0 to sizeof (WireSigSet) do
2:   if TST[stateIndex.x][yIndex].EnabledFlag == true then
3:     push (dynamicSet, WireSigSet[yIndex]);
4:   end if
5: end for
6: return dynamicSet;
```

Figure 4.19. Algorithm for generating dynamic set.

```
Procedure GenPOD (befIndex, aftIndex, TST, UserDefPOD);
1: befCausalArray ← GenCausal (TST, befIndex);
2: aftCausalArray ← GenCausal (TST, aftIndex);
3: POD ← MatchPOD (befCausalArray, aftCausalArray, UserDefPOD);
4: return POD;
```

Figure 4.20. Algorithm for generating point-of-divergence.

```
Procedure GenCausal (TST, TransIndex);
 1: for xIndex = TransIndex.x to 0 do
 2:   if TST[xIndex][TransIndex.y].Enabled == true then
 3:     if xIndex == 0 then
 4:       break;
 5:     else
 6:       continue;
 7:     end if
 8:   else
 9:     newIndex.x ← xIndex;
10:     newIndex.y ← findYIndex (xIndex + 1);
11:     push_front (CausalArray, newIndex);
12:     GenCausal (TST, newIndex);
13:   end if
14: end for
15: return CausalArray;
```

Figure 4.21. Algorithm for generating full causal list of transitions.

```
Procedure MatchPOD (befCausalArrary, aftCausalArray,
                    UserDefPOD);
 1: for index = 0 to min(sizeof(befCausalArray), sizeof(aftCausalArray)) do
 2:    if befCausalArrary[index] == aftCausalArray[index] then
 3:       push_front(commonArray, befCausalArray[index]);
 4:    else
 5:       break;
 6:    end if
 7: end for
 8: if UserDefPOD is empty then
 9:    POD ← commonArray.last();
10: else
11:    if UserDefPOD ∈ commonArray then
12:       POD ← UserDefPOD;
13:    else
14:       report_error();
15:       exit();
16:    end if
17: end if
```

Figure 4.22. Algorithm for matching POD.

Table 4.1. An example of trace status table.

	W	0	1	2	3	4	5
0	a	A00,0,T,F	A01,1,F,F	A02,1,F,F	A02,1,F,F	A00,1,T,F	A01,2,F,F
1	b	B00,0,T,F	B01,0,T,F	B02,1,F,F	B02,1,F,F	B00,1,T,F	B01,1,T,F
2	c	C00,0,F,F	C05,0,F,F	C01,0,T,F	C06,1,F,F	C06,1,F,F	C02,1,F,F
3	d	D00,0,F,F	D05,0,F,F	D05,0,F,F	D05,0,F,F	D01,0,T,F	D01,0,T,T
4	e	E00,0,F,F	E02,0,F,F	E05,0,F,F	E05,0,F,F	E01,0,T,F	E01,0,T,F
5	d	F00,0,F,F	F03,0,F,F	F03,0,F,F	F01,0,T,F	F12,1,F,F	F12,1,F,F
	T	init	a	b	c	d	a

CHAPTER 5

CASE STUDY

This chapter studies some real examples ranging from a simple C-element, to linear controller and then to a relatively complex six-four GasP circuit to demonstrate how the algorithms described in Chapter 4 work for computation interference and nonconformant illegal output errors and the design flow described in Chapter 2.

5.1 Simple C-element

A C-element is a commonly used element in asynchronous circuit design. It outputs goes low if its two inputs are both low whereas the output goes high if its two inputs are both high. For other combinations of input values, the output retains its previous value. Figure 5.1 and Table 5.1 are the symbol and truth table of the C-element, respectively.

Figure 5.2 is a C-element implemented with three 2-input and one 3-input NAND gates. This is a simple enough example to demonstrate how the algorithms work on a computation interference error. Based on the observable input and output behavior of C-element, its specification in CCS can be modeled as

$$\text{CSPEC} = \text{a.b.}\overline{c}\text{.CSPEC} + \text{b.a.}\overline{c}\text{.CSPEC}$$

and its CCS circuit specification is shown in Figure 5.3.

The CCS specification of the C-element design is composed of four elements using the parallel composition operator "|" such that each agent evolves concurrently. Lines 2 to 5 define the four primitive NAND gates with their initial state and the input and output port mappings. As an example, line 2 represents the NAND gate whose output is signal `ab`. NAND001 is its initial state where the first 0 denotes the input port `a` to be logically low, the second 0 denotes the input port `b` to be logically low as well, and the last 1 denotes the output port `c` is initially high. The naming convention

of initial values of input and output ports follows the rule by which output ports use 0 and 1 to represent logical low and high while input ports uses 0 and port names to represent logical low and high. NANDabc0 means that the initial values of input ports are all 1s and output port is low. The input and output port names of local components are relabeled by the real connection wire names using relabeling operator "/", e.g. ab/a means port a is relabeled with wire name ab. Line 6 uses restriction operation "\" to abstract away the internal signals.

Let us take an error trace as an example to demonstrate how to solve a computation interference error. The trace returned by the formal verification engine is as `a b ab c a` and its logic level and unrolling count mapping is shown in Table 5.2. The mapping from CCS to logic levels depends on the initial values of the signals. For the C-element shown in Figure 5.2 the initial value of `a`, `b`, `c` is 0 whereas the initial value of internal signal `ab`, `ac` and `bc` is 1. The unrolling count representation of signal transition just counts the number of times a signal has changed. Note that the first transition is denoted as unrolling count 0 instead of 1.

The trace status tableau is then constructed as in Table 5.3. The bottom row is the error trace, the second leftmost column lists the complete set of wire node signals. Signals `a` and `b` are primary inputs, signal `c` is the primary output, and `ab`, `bc`, `ac` are internal signals. The horizontal and vertical indices are explicitly specified for illustration purpose. The states in the cells of trace status tableau are simplified states due to width limitations of the paper. The primary inputs `a` and `b` share one state graph of the specification denoted as S. The internal signals and the primary output have their own state graphs associated with each gate.

In this computation interference example, `a` 1 (`a`−) is the failure transition. The *Failed* flag is true in the cell of TST[5][3] indicating that the POC is `ac` and the error occurs at gate B. The current state is B01, which is actually the state NANDab1 in line 4 of Figure 3.3. The only transition signal gate B can make is to fire its output `ac`. The failure transition `a` 1 is trying to disable the output transition and causes computation interference. The information used for generating relative ordering is summarized as follows.

- α_{fail}: `a` 1 (`a`−)

- Current state: $I' = (A06|B01|C01|D12), S' = S00$
- Previous state: $I = (A06|B05|C05|D01), S = S02$
- α_{en}: c 0 (c+)
- **dynamic**(I'): b 1 (b−), ac 0 (ac−), bc 0 (bc−)
- **dynamic**(I): c 0 (c+)

The state of POC changes from B05 to B01 by transition c 0 (c+). The **dynamic** sets at C01 and C05 can be generated by traversing column 5 and 3 respectively to find if the *Enabled* flag is true. All of the information is mapped to a state transition graph shown in Figure 5.4.

There is no available candidate relative ordering from state B05 since the only enabled and ready-to-fire transition is c+. At state B01, three enabled transitions of the system can fire before α_{fail}.

1. b 1 $\prec$ a 1; (b− $\prec$ a−)
2. ac 0 $\prec$ a 1; (ac− $\prec$ a−)
3. bc 0 $\prec$ a 1; (bc− $\prec$ a−)

The above set of relative timing constraints contains nonPOC signals since b and ac do not belong to the **sort** set of the POC gate B. If the option of POC constraint is enabled, only bc 0 $\prec$ a 1 will be returned.

The algorithms are able to remove the failures but do not evaluate if the candidate relative timing constraints generated may lead to other errors. For example, the candidate relative ordering b− $\prec$ a− is a bad constraint because the b− transition in Figure 5.4 actually leads to a failure state as well.

Let us take relative ordering bc 0 $\prec$ a 1 as an example to demonstrate how to generate the point-of-divergence. Transition bc 0 is enabled by transition c 0 by backtracking *Enabled* flag of row 4. c 0 is enabled by transition ab 0 by backtracking *Enabled* flag of row 5. Finally ab 0 is enabled by b 0 by backtracking *Enabled* flag of row 2. Likewise, the full causal path of transition a 1 can be backtracked in the

same way. The full causal paths of both events are listed in Table 5.4. Transition `b` 0, `ab` 0, `c` 0 can all be points-of-divergence. By default `c` 0 is returned. If the user specifies `b` as his/her desired POD, `b` 0 is returned.

There are a total of 25 sets of relative timing constraints shown in Table 5.5 that can be applied to the circuit implementation such that the circuit conforms to the specification to become hazard-free. The solution sets of constraints are unoptimized constraints and the number varies from 4 to 6. The difference in the number of constraints is affected by the strength of a constraint which will be presented in Section 6.2. Non-POC constraints are normally stronger than strict POC constraints. The constraints of 200 – 203 are all non-POC constraints and result in the most compact set.

A single error may have one or more relative timing solution constraints. Since the solution constraints for a single error are mutually exclusive, each time only one of them is added to the current available set of constraints and triggers another run of formal verification. To guarantee the completeness of analysis, all the relative timing constraint combinations are evaluated. Thus the relative timing constraints grow like a tree. Every node of the constraint tree represents an error type whereas every arc represents a relative timing constraint. More intuitively, at a node where a computation interference occurs, its egress arcs represent the relative timing constraints that solve this error. If it is an unsolvable error such as deadlock or some input/output transition is required but not possible to be generated by the circuit implementation, this node is marked as bad. If no error occurs after a set of constraints is applied, the node is marked as good. Every path from root node to a good node is a solution set of relative timing constraints.

Options can be specified by the user to deliver his/her preference on how the tool generates relative timing constraints. A set of solution constraints can be *quickly* generated by employing a depth first method. This method always chooses one constraint path and continues running verification until there exist no errors. The set of constraints generated by this option may have more constraints but uses less time and resources. Another option employs breadth first search method where every possible constraint must be evaluated before the next level run starts. This always

results in the most compact set of solution constraints but consumes more resources because every auxiliary paths in the tree must be stored. The user can also specify an option to return all the possible solution sets of relative timing constraints. Figure 5.5 shows a tree of relative timing constraints. The black nodes are bad nodes while pure white nodes are good nodes. The nodes with "ci" means that it is a computation interference error. {rt0, rt01}, {rt0, rt03}, {rt2, rt20} and {rt2, rt21} are solution sets of relative timing constraints.

5.2 Six-Four GasP Circuit

5.2.1 Introduction to GasP

The GasP family of asynchronous circuits shows ultra high speed by transporting data either in linear pipeline or switch fabric. The 4-2 GasP circuit operates at the speed of a three-inverter ring oscillator, and a test chip in 0.35μ technology exhibits throughput of 1.5 giga data items per second (GDI/s) [20]. The GasP family circuits, including basic, branch and merge modules, are used in an experimental test chip $FLEET_{zero}$ [62]. This work employs a completely different architecture than traditional op code based designs and emphasizes a communication centric paradigm. Data transforming is performed at local *ship* elements and data transporting is performed by the only instruction *move* through a switch fabric which is built with GasP family circuits.

GasP circuits employ single-track handshake signaling [63, 64, 65]. Namely a single wire functions as both request and acknowledge of handshake protocol. Figure 5.6 shows the circuit diagram of a 6-4 basic GasP circuit [66]. The `pred` and `succ` signals represent state wires of predecessor and successor pipeline stages. The logical low and high levels of `pred` and `succ` indicate their corresponding predecessor and successor stages are full and empty respectively. The `fire` signal makes the data path latch transparent if the predecessor is detected full and the successor is detected empty. The `fire` signal then resets the predecessor to be empty and sets the successor to be full. It takes six gate delays of forward latency from the predecessor stage being full to successor stage being full and four gate delays of backward latency from a successor stage being empty to its predecessor stage being empty. This is the reason

why it is called a 6-4 GasP circuit. Likewise, the 4-2 and 4-4 GasP circuits follow the same naming rule with different forward and backward latencies.

The relative timing constraints of 6-4 GasP circuit has been validated with postlay-out extracted parasitics using clocked static timing analysis engine in [67]. However, these relative timing constraints are generated by hand from the intuition only and may not be a full set of relative timing constraints.

5.2.2 Converting Single Track to Double Track

Single track signaling was first proposed by van Berkel to combine the advantages of two-phase and four-phase handshaking protocols by employing a single wire for both request and acknowledge signaling [63]. It requires only one wire and two transitions to complete a handshake cycle. The state wire also returns to its initial logic level when a handshaking is done.

The formal verification engine employed in this thesis does not directly support single track signaling. Fortunately, the symmetry of the GasP family circuit structure allows us to re-partition a pipelined GasP circuits into a double track handshake protocol. Figure 5.7 shows a 3 deep GasP pipeline with the new partition. This is a linear pipeline composed of three 6-4 basic GasP circuits that is delimited by dashed lines and named by lower case letters. The partitioned double track GasP circuits are delimited by double dotted lines and named by capital letters.

This repartitioning process changes only the hierarchy of the pipeline and the logic remains unchanged. The GasP family also includes branch and merge modules upon which switch fabrics can be built. The branch and merge modules follow the same structure as the basic module by resetting predecessors with stand-alone n-mos transistor and setting the successor with stand-alone p-mos transistors. This makes it a perfectly seamless partition. Figure 5.8 is a simplified switch network that is composed of basic, branch and merge modules. Whenever there is a request from a previous stage, the branch module chooses the destination path from paralleled pipelines based on the direction bit. The merge module simply receives one request one time from two paths and passes it to the next stage. Due to the page margin, only one pipe stage that uses a basic GasP module is drawn.

The repartitioned GasP module "splits" the single track handshaking into separate request and acknowledge signals (shown in Figure 5.9) which fit the formal verification engine very well. This hierarchy cut requires that a single diffusion connected network (DCN) to be split across the protocol channels. To allow correct modeling of the gate, the complementary dynamic gates connected to the same wire must be modeled as a single function. Thus the single p-mos pull-up and single n-mos pull-down logic, together with two serial p-mos pull-up and two serial n-mos pull-down keepers, can be modeled as a diffusion connected network (DCN) gate for speed-independent verification since wire delay is not considered in this case.

The repartition moves the long state wire between original GasP pipeline inside the new hierarchical module. This long wire plays an important role of performance of GasP pipelines [68]. The wire delay must be taken into account and delay insensitive verification is imperative.

To perform delay insensitive verification, all the wire forks must be modeled into gates such that the unbounded delay is assumed. There are two ways to model wire forks.

- Add one-to-two fork module at each node where a wire fork exists.
- Add buffers to all the branching out paths.

The CCS definition of a one-to-two fork module is defined as

$$\text{FORK} = \text{a.('b.'c.FORK + 'c.'b.FORK)}$$

If there exists a one-to-multiple fork node, two or more one-to-two fork modules will be used. The buffers can be added to all the branches to model arbitrary ordering of occurrences as well. Normally when a wire fork has only two branches and is branching to the same gate, one-to-two fork modules will be used. When it is a one-to-multiple fork and some of the branches already have single input single output elements such as buffers and inverters, adding buffers to those branches that directly outputs to a multi-input gates is preferred. In other cases, the designer can use either method based on his/her preference. Figure 5.10 shows the double track GasP basic module with fork module or buffers added. The squares represent one-to-2 fork modules while triangles without bubbles represent buffers.

The node `lo` in Figure 5.9 is actually pulled up by predecessor and pulled down by successor. Putting both pull-up and pull-down logic in a single module may mislead the reader. If the node `lo` is pulled up, the transition `lo+` will pass through the long wire to reach the successor `lo_1` and `lo_0` while `lo_2` belongs to the predecessor and can see the pull-up immediately. Likewise `lo_1` and `lo_0` will see the pull-down sooner than `lo_2`. Hence pull-up and pull-down directions both need buffers to model the long wire delay. Since an inverter of `lo_0` is already there, a buffer is added to the output of `lo`.

Now that the double track new GasP architecture satisfies the requirements of the formal verification engine, model checking and relative timing constraint generation by ARTIST can be performed. Figure 5.11 is the CCS specification of the behavior of 6-4 basic GasP and Figure 5.12 is the CCS specification of the circuit implementation for speed-independent verification where FC0Iabc00 is the DCN gate. This is a relatively loose specification with less concurrency. The GasP family circuits make use of wire delays to allow transient short circuits and achieve as much concurrency as possible. However the DCN gate is modeled to be able to detect and report short circuit failures which restricts the use of a more concurrency specification.

Table 5.6 lists one candidate set of relative timing constraints. This set has a total of 11 relative timing constraints that have been optimized such that redundant constraints have been removed. The set of constraints is generated by a fully automatic verification run that sets the depth first option.

Each relative timing constraint in Table 5.6 is explained in detail as follows. The error trace is listed both with CCS and logic level formats. The causal paths from point-of-divergence to point-of-convergence are listed as well. The unit gate delay for each path is listed to show the number of signals that switch in the race paths to determine if the constrained timing seems reasonable. (The early path should be shorter than the late arriving path.) Each relative timing constraint is associated with a circuit diagram figure with racing paths specified where blue arrow represents shorter path and red arrow represents longer path. Finally the errors and their corresponding constraints are analyzed from the perspective of the formal verification engine and ARTIST as well as the logic view of the GasP circuit.

- rtc = rt0 : lo 0 ⇒ lo_2 0 ≺ li 1; (rtc = rt0 : lo+ ⇒ lo_2- ≺ li+;)
  - **Error Trace (CCS)**: li 'lo li 'lo_1
  - **Error Trace (+/-)**: li- lo+ li+ lo_1-
  - **PATH$_{\text{pod-poc0}}$**: lo+ lo_2-
  - **PATH$_{\text{pod-poc1}}$**: lo+ li+
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 4
  - **Description**: Short circuit failure on pull-up and pull-down keepers. The second transition of `li` (`li+`) makes pull-down keeper enabled. `lo_1−` will enable pull-up keeper and cause short circuit. One of the solutions is to force `lo_2−` to disable the pull-down keeper before signal `li` is reset. It is more straightforward to enforce the ordering `lo_2−` ≺ `lo_10` to disable pull-down keeper and then enable pull-up keeper (this fits the long state wire delay in reality). But constraint rt0 is relatively stronger. The constraint satisfies the unit delay gate counts. See Figure 5.13.
- rtc = rt1 : lo 0 ⇒ li 1 ≺ fire 0;
  - **Error Trace (CCS)**: li 'lo 'lo_0 'chk 'chk_ 'fire
  - **Error Trace (+/-)**: li- lo+ lo_0- chk+ chk_- fire+
  - **PATH$_{\text{pod-poc0}}$**: lo+ li+
  - **PATH$_{\text{pod-poc1}}$**: lo+ lo_0- chk+ chk_- fire+
  - **Unit Gate Delay**: pod-poc0: 4; pod-poc1: 4
  - **Description**: This is a short circuit failure on functional p-mos and n-mos transistors. Transition `fire+`, which makes the latch transparent, also sets the predecessor to be empty by enabling pull-down p transistor. Thus disabling pull-up by `li−` before triggering pull-down by `fire+` is reasonable solution. Note that both paths from point-of-divergence to point-of-convergence have the same gate delays as 4. However, the path to `fire+` passes through the long wire between GasP pipeline. Hence the relative timing constraint still satisfies unit gate delay paradigm in the

real circuit although wire delay is not considered in this speed independent verification. See Figure 5.14.

- rtc = rt2 : lo 0 $\Rightarrow$ lo_1 0 $\prec$ lo 1;
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_0 'chk 'chk_ 'fire 'lo
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_0- chk+ chk_- fire+ lo-
  - **PATH$_{\text{pod-poc0}}$**: lo+ lo_1-
  - **PATH$_{\text{pod-poc1}}$**: lo+ lo_0- chk+ chk_- fire+ lo-
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 5
  - **Description**: This is a typical computation interference error where the input is trying to disable an output transition. The output of inverter `lo_1`− does not fire even when a new input transition `lo`− comes in. From the function view of the circuit, `lo_1`− should occur such that the pull-up keeper is enabled. From the unit delay view, it is obvious that `lo_1`− should occur before `lo`−. See Figure 5.15.
- rtc = rt3 : fire 0 $\Rightarrow$ lo 1 $\prec$ fire 1;
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_0 'chk 'chk_ 'fire 'ro ri 'chk 'chk_ 'fire
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_0- chk+ chk_- fire+ ro- ri+ chk- chk_+ fire-
  - **PATH$_{\text{pod-poc0}}$**: fire+ lo-
  - **PATH$_{\text{pod-poc1}}$**: fire+ ro- ri+ chk- chk_+ fire-
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 5
  - **Description**: This is the same with rt2 as a computation interference error. The transition `fire`+ is supposed to reset predecessor to be by pulling down `lo`. However, before `lo`− fires, `fire`+ sets the successor to be full and then `fire` is reset to be low and trying to disable `lo`−. Firing `lo`− anyway before `fire`− solves the error. Apparently the constraint satisfies unit gate delay counts. See Figure 5.16. This is the same with

rt2 as a computation interference error. The transition `fire+` is supposed to reset predecessor to be by pulling down `lo`. However, before `lo−` fires, `fire+` sets the successor to be full and then `fire` is reset to be low and trying to disable `lo−`. Firing `lo−` anyway before `fire−` solves the error. Apparently the constraint satisfies unit gate delay counts. See Figure 5.16.

- rtc = rt4: lo 1 $\Rightarrow$ lo_2 1 $\prec$ lo 2;
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro ri 'lo 'lo_1 'lo_0 'chk 'chk_ 'fire li 'lo
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ ro- ri+ lo- lo_1+ lo_0+ chk- chk_+ fire- li- lo+
  - **PATH$_{\text{pod-poc0}}$**: lo- lo_2+
  - **PATH$_{\text{pod-poc1}}$**: lo- li- lo+
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 5
  - **Description**: This is a computation interference failure which is similar to rt3. However, this time it is pull-down operation. Transition `lo−` should turn on the pull-down keeper by `lo_2+` but before it fires, empty status is issued and another new request comes in and consequently tries to disable `lo_2+`. Firing `lo_2+` anyway solves the error. The constraint holds based on unit gate delay counts. See Figure 5.17.
- rtc = rt5 : lo 1 $\Rightarrow$ lo_1 1 $\prec$ fire 1;
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro ri 'lo 'lo_2 'chk 'chk_ 'fire
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ ro- ri+ lo- lo_2+ chk- chk_+ fire-
  - **PATH$_{\text{pod-poc0}}$**: lo- lo_1+
  - **PATH$_{\text{pod-poc1}}$**: lo- lo_0+ chk- chk_+ fire-
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 4

  - **Description**: Short circuit failure occurs in keeper pull-up and pull-down stacks. It is similar with rt0. The firing of `fire−` enables pull-up keeper and causes short circuit failure. One of the solutions is to force `lo_1+` to disable pull-up keeper before `fire` is reset. It is more straightforward to enforce the ordering `lo_1+` $\prec$ `lo_2+` to disable pull-up keeper and then enable pull-down keeper (this fits the long wire delay in reality as well). But constraint rt5 is relatively stronger. This is a perfect symmetry with rt0. See Figure 5.18.

- rtc = rt6 : fire 0 $\Rightarrow$ ro 0 $\prec$ fire 1;

  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'lo 'lo_1 'lo_0 'chk 'chk_ 'fire
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ lo- lo_1+ lo_0+ chk- chk_+ fire-
  - **PATH$_{\text{pod-poc0}}$**: fire+ ro-
  - **PATH$_{\text{pod-poc1}}$**: fire+ lo- lo_0+ chk- chk_+ fire-
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 5
  - **Description**: The is the same with rt2 as a computation interference error. Transition `fire+` is supposed to set the successor to be full. However, before `ro−` fires, `fire+` resets the predecessor state wire to be empty and then resets itself, thus trying to disable `ro−`. Firing `ro−` anyway before `fire−` solves the error. The constraint matches the unit gate delay counts. See Figure 5.19.

- rtc = rt7 : fire 0 $\Rightarrow$ ri 0 $\prec$ ro 1;

  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro 'lo 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ ro- lo- lo_1+ lo_0+ chk- chk_+ fire- ro+
  - **PATH$_{\text{pod-poc0}}$**: fire+ ro- ri+

- **PATH$_{\text{pod-poc1}}$**: fire+ lo- lo_0+ chk- chk_+ fire- ro+
- **Unit Gate Delay**: pod-poc0: 2; pod-poc1: 6
- **Description**: This is a nonconformance error. An illegal output `ro`+ is encountered. Simply force `ri`+ to occur before `ro`+ as indicated in the specification to avoid the failure. This is actually caused by the fact that `fire`+ resets predecessor to be empty and then resets itself in a faster way than other path that sets successor to be full. This failure is a polymorphism of rt6 with additional firing `ro`−. This constraint is not strict POC constraint where the two events are not converged to a single point. A much stronger strict POC constraint is `fire`+ $\Rightarrow$ `ri`+ $\prec$ `lo_0`+. The constraint satisfies the unit gate delay counts. See Figure 5.20.

- rtc = rt8 : lo 1 $\Rightarrow$ lo_0 1 $\prec$ lo 2;
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro ri 'lo 'lo_2 'lo_1 'chk 'chk_ 'fire li 'lo
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ ro- ri+ lo- lo_2+ lo_1+ chk- chk_+ fire- li- lo+
  - **PATH$_{\text{pod-poc0}}$**: lo- lo_0+
  - **PATH$_{\text{pod-poc1}}$**: lo- li- lo+
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 5
  - **Description**: The is the same with rt2 as a computation interference error. After the predecessor is reset to be empty, it is filled with a new data. But the reset path of `fire` still does not proceed (in this case, it is reset by another path through `ro`− and `ri`+). Transition `lo`+ is trying to disable `lo_0`+. It is obvious that the constraint satisfies unit gate delay counts. See Figure 5.21.

- rtc = rt9 : fire 0 $\Rightarrow$ lo_0 1 $\prec$ ri 1;
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro ri 'lo 'lo_1 'chk 'chk_ 'fire 'ro ri 'lo_0

- **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ ro- ri+ lo- lo_1+ chk- chk_+ fire- ro+ ri- lo_0+
- **PATH$_{\text{pod-poc0}}$**: fire+ lo- lo_0+
- **PATH$_{\text{pod-poc1}}$**: fire+ ro- ri+ chk- chk_+ fire- ro+ ri-
- **Unit Gate Delay**: pod-poc0: 2; pod-poc1: 7
- **Description**: Transition `fire`+ should reset predecessor to be empty and successor to be full. Signal `ri` goes high and then low means that the data element in successor has already been sent out to its next stage. However, when the NOR gate detects the state wires of predecessor and successor at the moment `ri`−, it finds out that the predecessor is still full since `lo_0`+ does not fire yet. This incorrect detection will enable `fire` again. Fortunately this error embodies a computation interference in this case where `lo_0`+ is trying to disable NOR gate which is enabled by `ri`−. Moreover the unit gate delay counts indicate that the delay of path from `fire`+ to `lo_0`+ is less than the time the successor consumes and sends out the data. Hence firing `lo_0`+ before `ri`+ correctly updates the predecessor's status and solves the error. See Figure 5.22.

- rtc = rt10: lo 2 $\Rightarrow$ lo_2 2 $\prec$ li 3; (rtc = rt10: lo+ $\Rightarrow$ lo_2- $\prec$ li+;)
  - **Error Trace (CCS)**: li 'lo 'lo_2 li 'lo_1 'lo_0 'chk 'chk_ 'fire 'ro ri 'lo 'lo_2 'lo_1 'lo_0 'chk 'chk_ 'fire li 'lo li 'lo
  - **Error Trace (+/-)**: li- lo+ lo_2- li+ lo_1- lo_0- chk+ chk_- fire+ ro- ri+ lo- lo_2+ lo_1+ lo_0+ chk- chk_+ fire- li- lo+ li+ lo-
  - **PATH$_{\text{pod-poc0}}$**: lo+(2) lo_2-(2)
  - **PATH$_{\text{pod-poc1}}$**: lo+(2) li+(3)
  - **Unit Gate Delay**: pod-poc0: 1; pod-poc1: 4
  - **Description**: This is a nonconformance error with illegal output `lo`. In this case the pull-down keeper performs the functional pull-down with `li`+ and `lo_2`+. The constraint fires `lo_2`− to disable pull-down keeper. The

logic level representation of rt10 is the same with rt0. This is a multicycle constraint. See Figure 5.13.

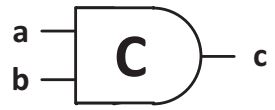


Figure 5.1. C-element symbol.

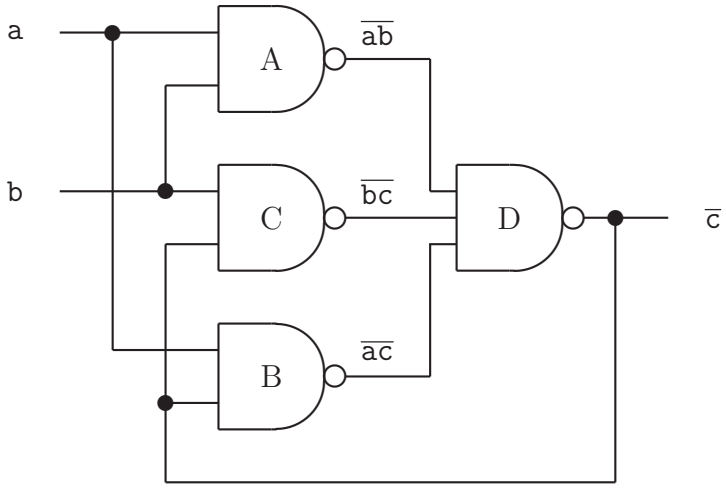


Figure 5.2. C-element implemented with three 2-input and one 3-input NAND gates.

```
1: C-ELEMENT =
2: (  NAND001[a/a, b/b, ab/c]
3:  | NAND001[a/a, c/b, ac/c]
4:  | NAND001[b/a, c/b, bc/c]
5:  | NANDabc0[ab/a, ac/b, bc/c, c/d]
6: ) \{ ab, ac, bc } ;
```

Figure 5.3. CCS implementation of C-element.

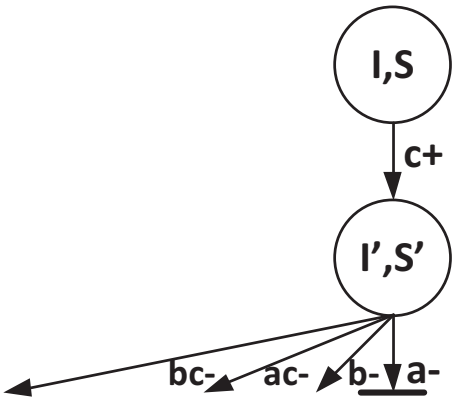


Figure 5.4. Partial state graph mapped from trace status tableau.

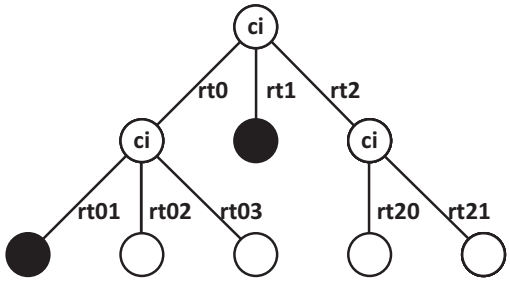


Figure 5.5. Tree of relative timing constraints.

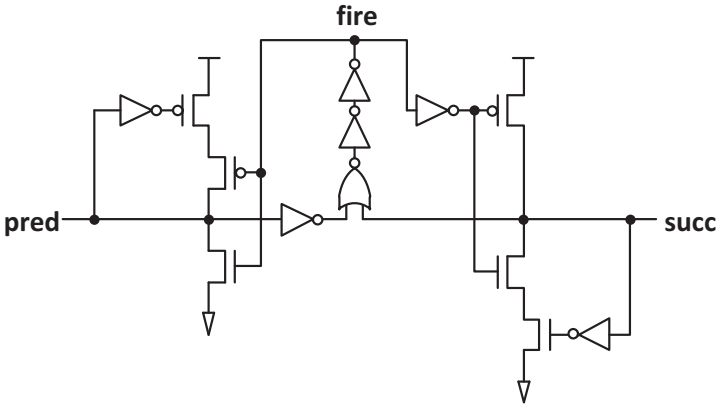


Figure 5.6. Six-Four basic GasP circuit.

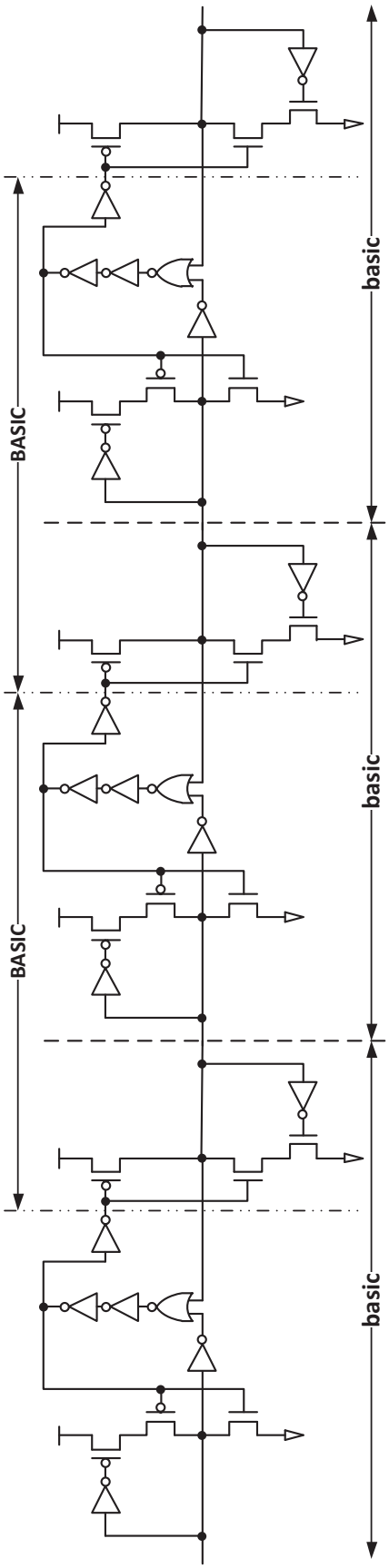


Figure 5.7. Repartition of 3 deep GasP pipeline.

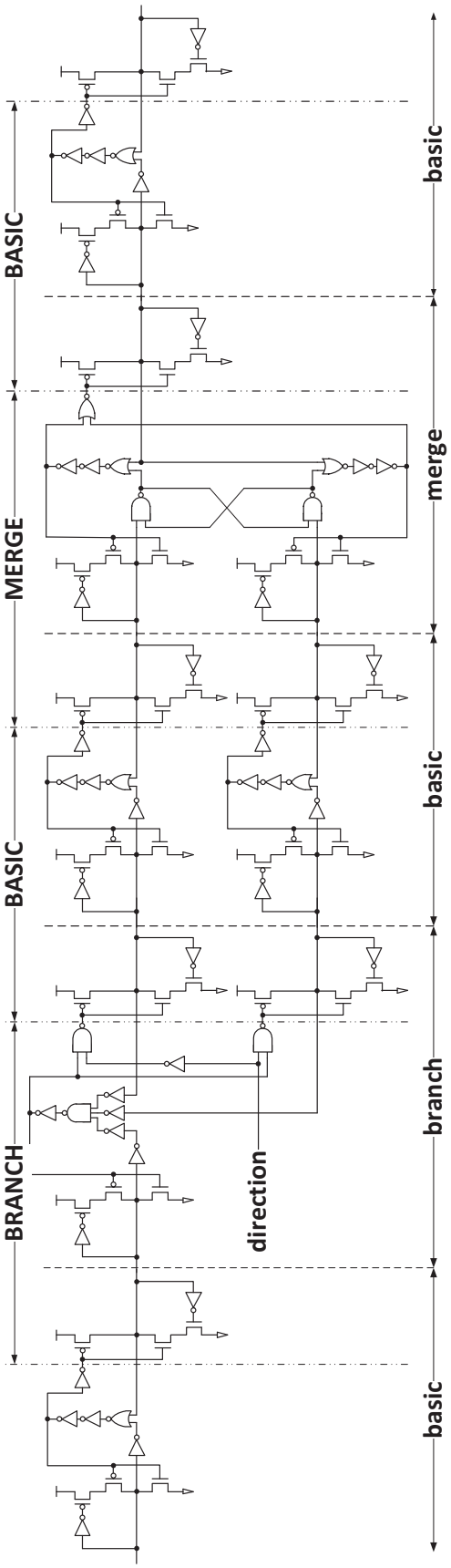


Figure 5.8. Repartition of a simplified switch network composed by basic, branch and merge GasP circuits.

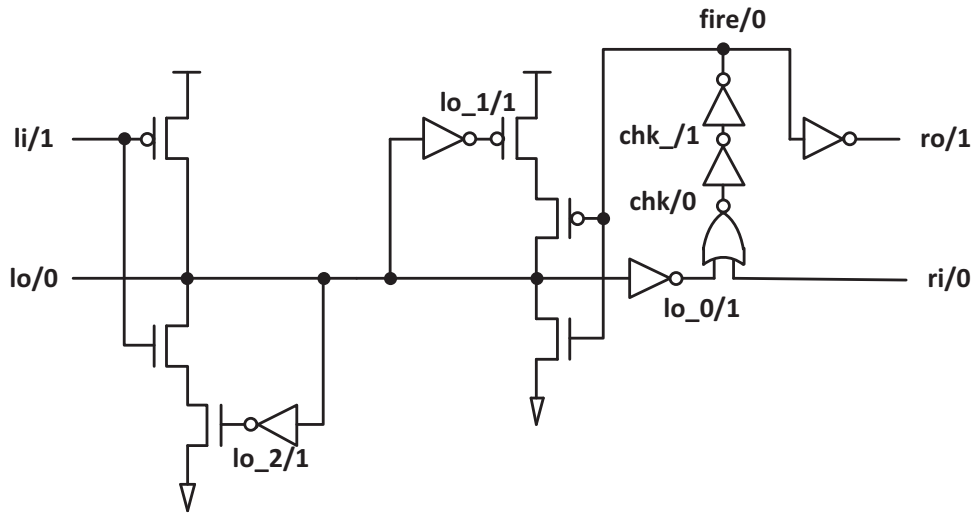


Figure 5.9. Speed-independent model of repartitioned double track GasP basic circuit.

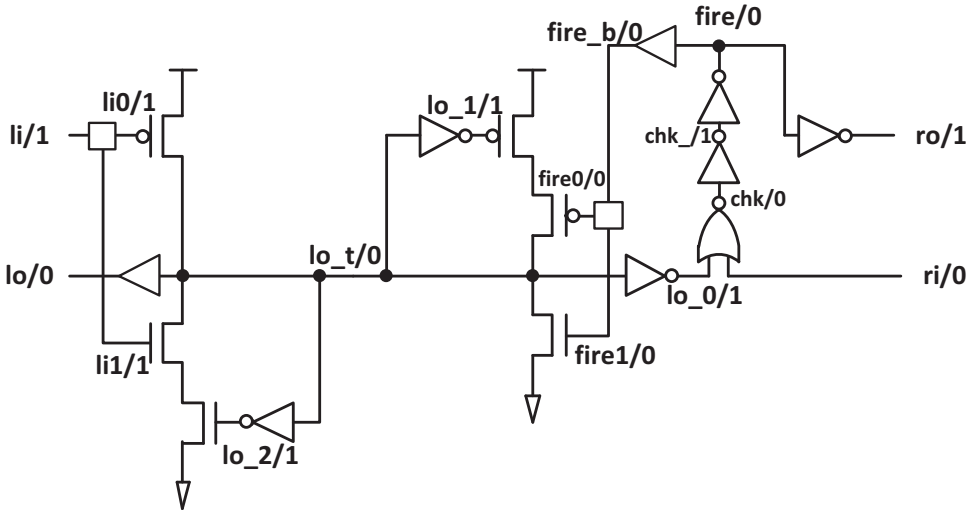


Figure 5.10. Delay-insensitive model of repartitioned double track GasP basic circuit.

```
L    =  li.'lo.lo.x.'lo.    x.       L
R    =           'x.'ro.ri.'x.'ro.ri.R
SPEC =  (L | R) \ {x}
```

Figure 5.11. Specification of double track GasP circuit.

```
GASPIMPL =
(  FC0Iabc00[li/a, lo_2/b, lo_1/c, fire/d, lo/e] \
 | INV01[lo/a, lo_0/b] \
 | INV01[lo/a, lo_1/b] \
 | INV01[lo/a, lo_2/b] \
 | NORa00[lo_0/a, ri/b, chk/c] \
 | INV01[chk/a, chk_/b] \
 | INVa0[chk_/a, fire/b] \
 | INV01[fire/a, ro/b] \
) \{ lo_0, lo_1, lo_2, chk, chk_, fire }
```

Figure 5.12. Speed-independent implementation of double track GasP circuit.

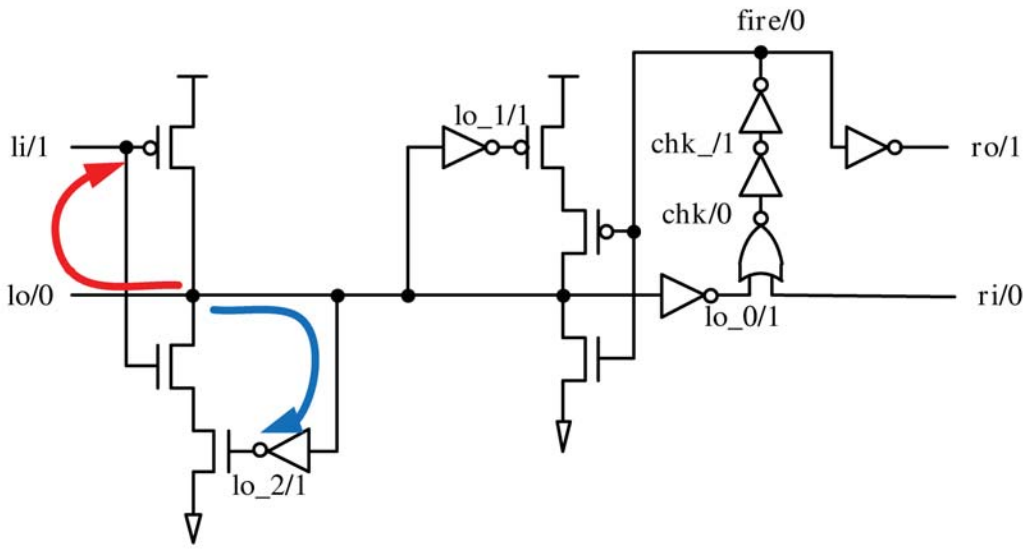


Figure 5.13. GasP speed-independent verification RT0.

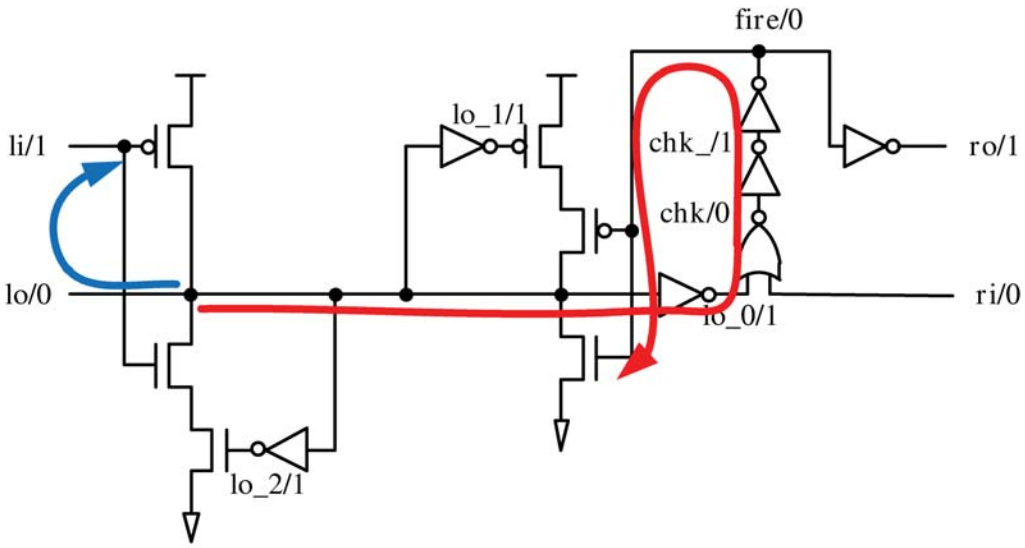


Figure 5.14. GasP speed-independent verification RT1.

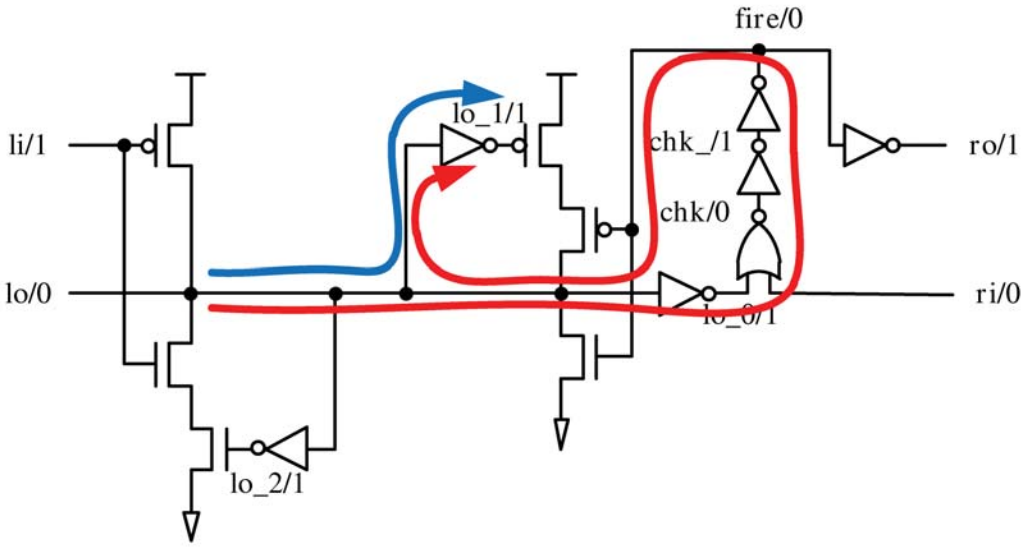


Figure 5.15. GasP speed-independent verification RT2.

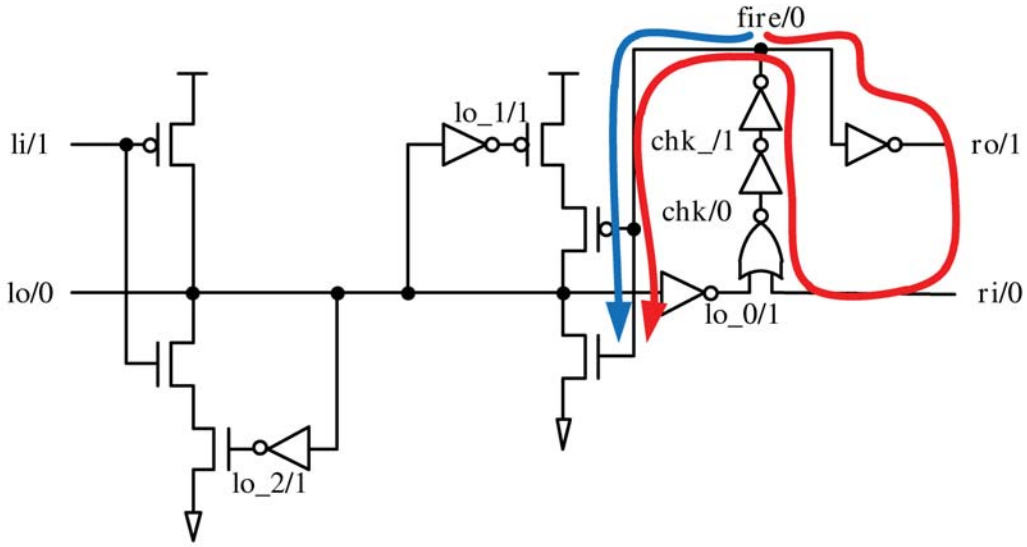


Figure 5.16. GasP speed-independent verification RT3.

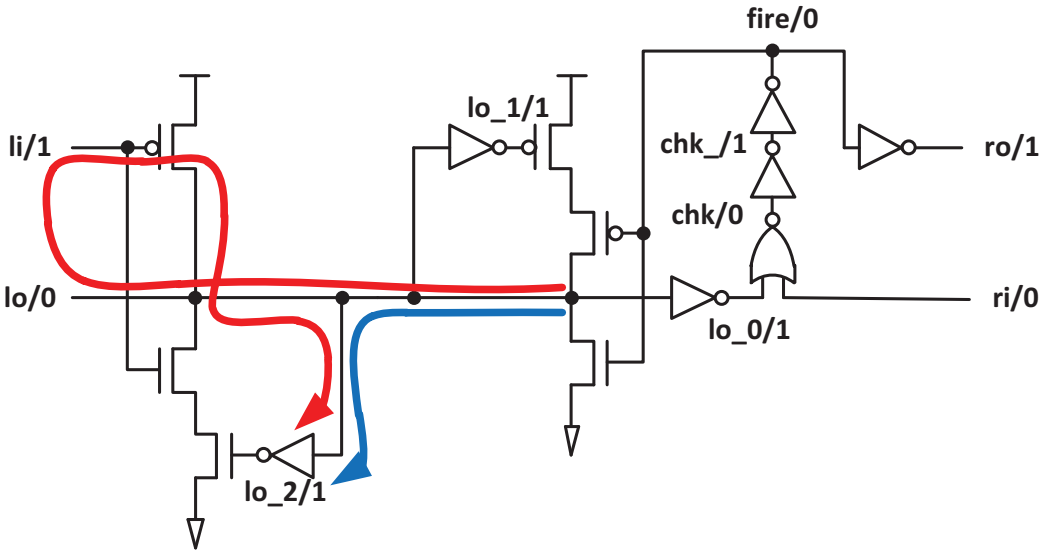


Figure 5.17. GasP speed-independent verification RT4.

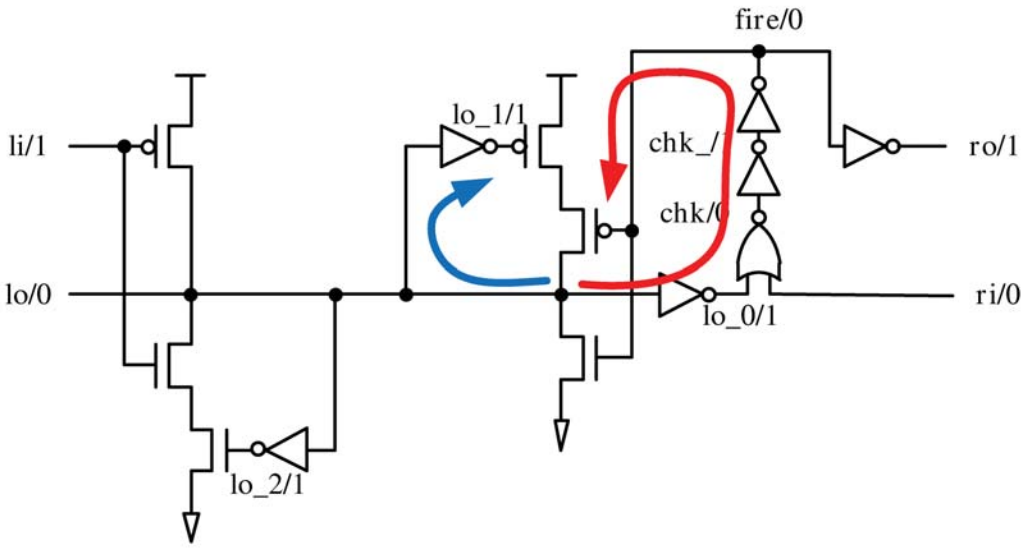


Figure 5.18. GasP speed-independent verification RT5.

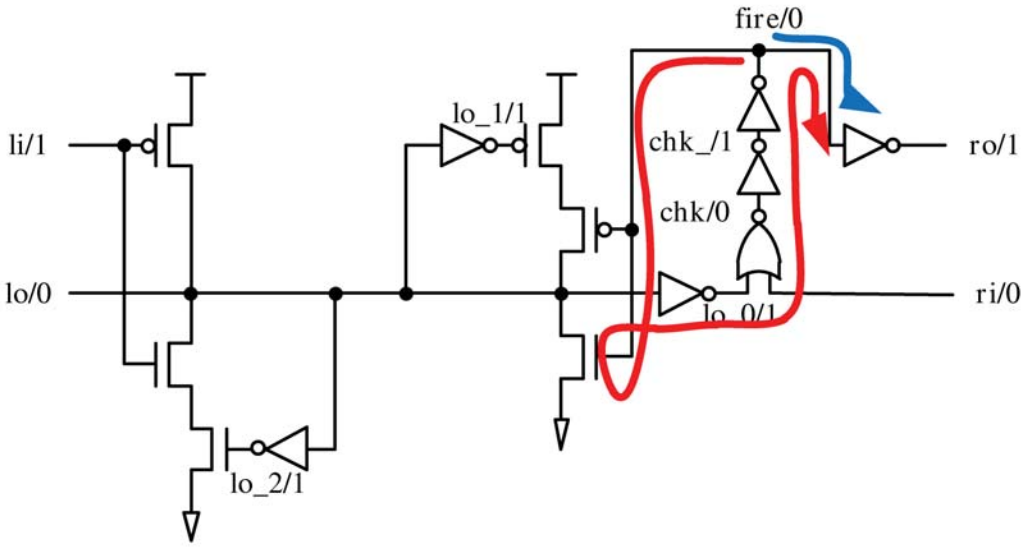


Figure 5.19. GasP speed-independent verification RT6.

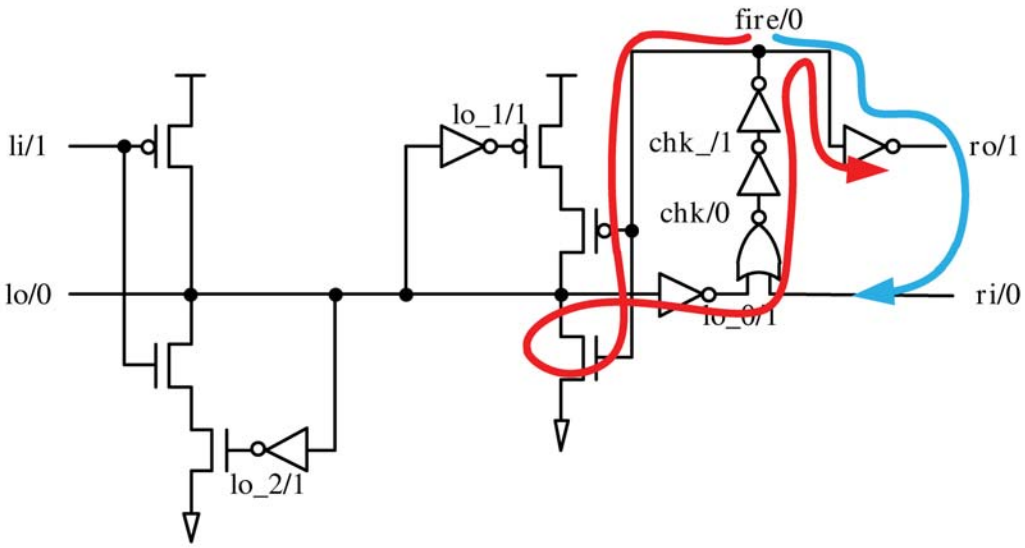


Figure 5.20. GasP speed-independent verification RT7.

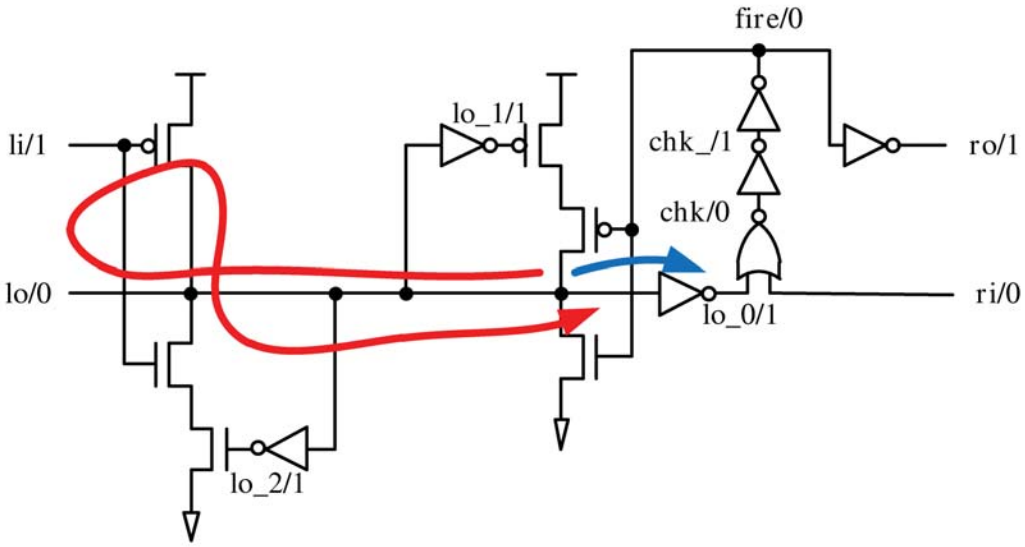


Figure 5.21. GasP speed-independent verification RT8.

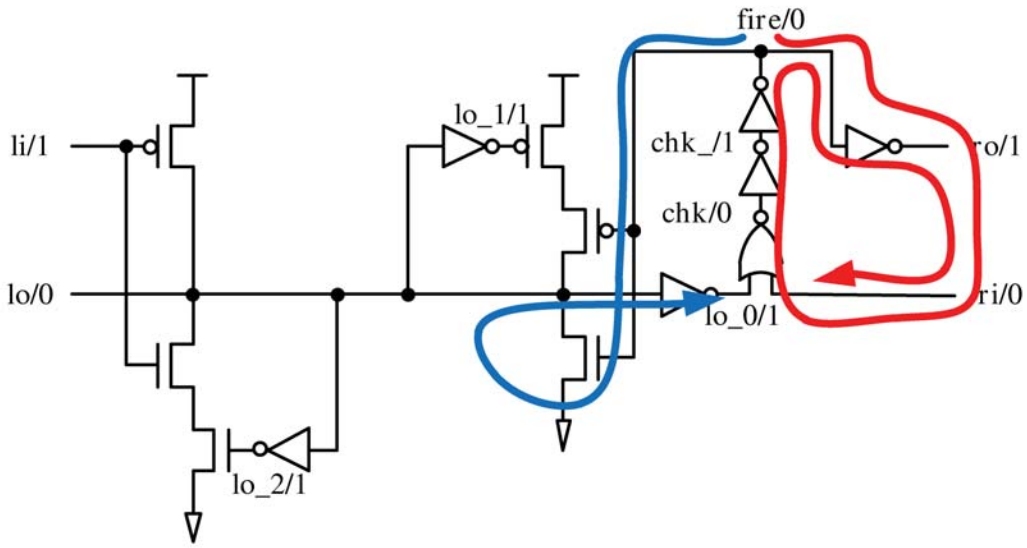


Figure 5.22. GasP speed-independent verification RT9.

Table 5.1. Truth table of C-element.

a	b	c
0	0	0
0	1	c_{n-1}
1	0	c_{n-1}
1	1	1

Table 5.2. Signal transition mapping of CCS, logic level and unrolling count representations.

CCS	a	b	ab	c	a
Logic Level	a+	b+	ab-	c+	a-
Unrolling	a 0	b 0	ab 0	c 0	a 1

Table 5.3. An example tableau for an error trace in verification of C-element.

TST	W	0	1	2	3	4	5
0	a	S00,0,T,F	S01,1,F,F	S02,1,F,F	S02,1,F,F	S00,1,T,F	S01,2,F,F
1	b	S00,0,T,F	S01,0,T,F	S02,1,F,F	S02,1,F,F	S00,1,T,F	S01,1,T,F
2	ab	A00,0,F,F	A05,0,F,F	A01,0,T,F	A06,1,F,F	A06,1,F,F	A02,1,T,F
3	ac	B00,0,F,F	B05,0,F,F	B05,0,F,F	B05,0,F,F	B01,0,T,F	B02,0,F,T
4	bc	C00,0,F,F	C02,0,F,F	C05,0,F,F	C05,0,F,F	C01,0,T,F	C01,0,T,F
5	c	D00,0,F,F	D03,0,F,F	D03,0,F,F	D01,0,T,F	D12,1,F,F	D12,1,F,F
	T	init	a 0	b 0	ab 0	c 0	a 1

Table 5.4. Full causal paths of relative ordering events.

Index	0	1	2	3
Shorter path	b 0	ab 0	c 0	**bc 0**
Longer path	b 0	ab 0	c 0	**a 1**

Table 5.5. Complete solution sets of RT constraints.

RT Constraints	RT Constraints	RT Constraints	RT Constraints	RT Constraints
rt000: c 0 ⇒ ac 0 ≺ a 1; rt001: c 0 ⇒ ac 0 ≺ b 1; rt002: c 0 ⇒ bc 0 ≺ b 1; rt003: c 0 ⇒ b 1 ≺ c 1; rt004: c 0 ⇒ b 1 ≺ ab 1;	rt010: c 0 ⇒ ac 0 ≺ a 1; rt011: c 0 ⇒ ac 0 ≺ b 1; rt012: c 0 ⇒ bc 0 ≺ b 1; rt013: c 0 ⇒ b 1 ≺ c 1; rt014: c 0 ⇒ bc 0 ≺ ab 1;	rt020: c 0 ⇒ ac 0 ≺ a 1; rt021: c 0 ⇒ ac 0 ≺ b 1; rt022: c 0 ⇒ bc 0 ≺ b 1; rt023: c 0 ⇒ bc 0 ≺ c 1; rt024: c 0 ⇒ b 1 ≺ ab 1;	rt030: c 0 ⇒ ac 0 ≺ a 1; rt031: c 0 ⇒ ac 0 ≺ b 1; rt032: c 0 ⇒ bc 0 ≺ b 1; rt033: c 0 ⇒ bc 0 ≺ c 1; rt034: c 0 ⇒ bc 0 ≺ ab 1;	rt040: c 0 ⇒ ac 0 ≺ a 1; rt041: c 0 ⇒ bc 0 ≺ b 1; rt042: c 0 ⇒ b 1 ≺ c 1; rt043: c 0 ⇒ b 1 ≺ ab 1; rt044: c 0 ⇒ a 1 ≺ c 1; rt045: c 0 ⇒ a 1 ≺ ab 1;
rt050: c 0 ⇒ ac 0 ≺ a 1; rt051: c 0 ⇒ bc 0 ≺ b 1; rt052: c 0 ⇒ b 1 ≺ c 1; rt053: c 0 ⇒ b 1 ≺ ab 1; rt054: c 0 ⇒ a 1 ≺ c 1; rt055: c 0 ⇒ ac 0 ≺ ab 1;	rt060: c 0 ⇒ ac 0 ≺ a 1; rt061: c 0 ⇒ bc 0 ≺ b 1; rt062: c 0 ⇒ b 1 ≺ c 1; rt063: c 0 ⇒ b 1 ≺ ab 1; rt064: c 0 ⇒ ac 0 ≺ c 1; rt065: c 0 ⇒ a 1 ≺ ab 1;	rt070: c 0 ⇒ ac 0 ≺ a 1; rt071: c 0 ⇒ bc 0 ≺ b 1; rt072: c 0 ⇒ b 1 ≺ c 1; rt073: c 0 ⇒ b 1 ≺ ab 1; rt074: c 0 ⇒ ac 0 ≺ c 1; rt075: c 0 ⇒ ac 0 ≺ ab 1;	rt080: c 0 ⇒ ac 0 ≺ a 1; rt081: c 0 ⇒ bc 0 ≺ b 1; rt082: c 0 ⇒ b 1 ≺ c 1; rt083: c 0 ⇒ bc 0 ≺ ab 1; rt084: c 0 ⇒ a 1 ≺ c 1; rt085: c 0 ⇒ a 1 ≺ ab 1;	rt090: c 0 ⇒ ac 0 ≺ a 1; rt091: c 0 ⇒ bc 0 ≺ b 1; rt092: c 0 ⇒ b 1 ≺ c 1; rt093: c 0 ⇒ bc 0 ≺ ab 1; rt094: c 0 ⇒ a 1 ≺ c 1; rt095: c 0 ⇒ ac 0 ≺ ab 1;
rt100: c 0 ⇒ ac 0 ≺ a 1; rt101: c 0 ⇒ bc 0 ≺ b 1; rt102: c 0 ⇒ b 1 ≺ c 1; rt103: c 0 ⇒ bc 0 ≺ ab 1; rt104: c 0 ⇒ ac 0 ≺ c 1; rt105: c 0 ⇒ a 1 ≺ ab 1;	rt110: c 0 ⇒ ac 0 ≺ a 1; rt111: c 0 ⇒ bc 0 ≺ b 1; rt112: c 0 ⇒ b 1 ≺ c 1; rt113: c 0 ⇒ bc 0 ≺ ab 1; rt114: c 0 ⇒ ac 0 ≺ c 1; rt115: c 0 ⇒ ac 0 ≺ ab 1;	rt120: c 0 ⇒ ac 0 ≺ a 1; rt121: c 0 ⇒ bc 0 ≺ b 1; rt122: c 0 ⇒ bc 0 ≺ c 1; rt123: c 0 ⇒ b 1 ≺ ab 1; rt124: c 0 ⇒ a 1 ≺ c 1; rt125: c 0 ⇒ a 1 ≺ ab 1;	rt130: c 0 ⇒ ac 0 ≺ a 1; rt131: c 0 ⇒ bc 0 ≺ b 1; rt132: c 0 ⇒ bc 0 ≺ c 1; rt133: c 0 ⇒ b 1 ≺ ab 1; rt134: c 0 ⇒ a 1 ≺ c 1; rt135: c 0 ⇒ ac 0 ≺ ab 1;	rt140: c 0 ⇒ ac 0 ≺ a 1; rt141: c 0 ⇒ bc 0 ≺ b 1; rt142: c 0 ⇒ bc 0 ≺ c 1; rt143: c 0 ⇒ b 1 ≺ ab 1; rt144: c 0 ⇒ ac 0 ≺ c 1; rt145: c 0 ⇒ a 1 ≺ ab 1;
rt150: c 0 ⇒ ac 0 ≺ a 1; rt151: c 0 ⇒ bc 0 ≺ b 1; rt152: c 0 ⇒ bc 0 ≺ c 1; rt153: c 0 ⇒ b 1 ≺ ab 1; rt154: c 0 ⇒ ac 0 ≺ c 1; rt155: c 0 ⇒ ac 0 ≺ ab 1;	rt160: c 0 ⇒ ac 0 ≺ a 1; rt161: c 0 ⇒ bc 0 ≺ b 1; rt162: c 0 ⇒ bc 0 ≺ c 1; rt163: c 0 ⇒ bc 0 ≺ ab 1; rt164: c 0 ⇒ a 1 ≺ c 1; rt165: c 0 ⇒ a 1 ≺ ab 1;	rt170: c 0 ⇒ ac 0 ≺ a 1; rt171: c 0 ⇒ bc 0 ≺ b 1; rt172: c 0 ⇒ bc 0 ≺ c 1; rt173: c 0 ⇒ bc 0 ≺ ab 1; rt174: c 0 ⇒ a 1 ≺ c 1; rt175: c 0 ⇒ ac 0 ≺ ab 1;	rt180: c 0 ⇒ ac 0 ≺ a 1; rt181: c 0 ⇒ bc 0 ≺ b 1; rt182: c 0 ⇒ bc 0 ≺ c 1; rt183: c 0 ⇒ bc 0 ≺ ab 1; rt184: c 0 ⇒ ac 0 ≺ c 1; rt185: c 0 ⇒ a 1 ≺ ab 1;	rt190: c 0 ⇒ ac 0 ≺ a 1; rt191: c 0 ⇒ bc 0 ≺ b 1; rt192: c 0 ⇒ bc 0 ≺ c 1; rt193: c 0 ⇒ bc 0 ≺ ab 1; rt194: c 0 ⇒ ac 0 ≺ c 1; rt195: c 0 ⇒ ac 0 ≺ ab 1;
rt200: c 0 ⇒ bc 0 ≺ a 1; rt201: c 0 ⇒ ac 0 ≺ b 1; rt202: c 0 ⇒ ac 0 ≺ a 1; rt203: c 0 ⇒ bc 0 ≺ b 1;	rt210: c 0 ⇒ bc 0 ≺ a 1; rt211: c 0 ⇒ bc 0 ≺ b 1; rt212: c 0 ⇒ ac 0 ≺ a 1; rt213: c 0 ⇒ a 1 ≺ c 1; rt214: c 0 ⇒ a 1 ≺ ab 1;	rt220: c 0 ⇒ bc 0 ≺ a 1; rt221: c 0 ⇒ bc 0 ≺ b 1; rt222: c 0 ⇒ ac 0 ≺ a 1; rt223: c 0 ⇒ a 1 ≺ c 1; rt224: c 0 ⇒ ac 0 ≺ ab 1;	rt230: c 0 ⇒ bc 0 ≺ a 1; rt231: c 0 ⇒ bc 0 ≺ b 1; rt232: c 0 ⇒ ac 0 ≺ a 1; rt233: c 0 ⇒ ac 0 ≺ c 1; rt234: c 0 ⇒ a 1 ≺ ab 1;	rt240: c 0 ⇒ bc 0 ≺ a 1; rt241: c 0 ⇒ bc 0 ≺ b 1; rt242: c 0 ⇒ ac 0 ≺ a 1; rt243: c 0 ⇒ ac 0 ≺ c 1; rt244: c 0 ⇒ ac 0 ≺ ab 1;

Table 5.6. Speed-independent set of RT constraints for 6-4 basic GasP circuit.

RT Constraints											
rtc	=	rt0	:	lo	0	$\Rightarrow$	lo_2	0	$\prec$	li	1
rtc	=	rt1	:	lo	0	$\Rightarrow$	li	1	$\prec$	fire	0
rtc	=	rt2	:	lo	0	$\Rightarrow$	lo_1	0	$\prec$	lo	1
rtc	=	rt3	:	fire	0	$\Rightarrow$	lo	1	$\prec$	fire	1
rtc	=	rt4	:	lo	1	$\Rightarrow$	lo_2	1	$\prec$	lo	2
rtc	=	rt5	:	lo	1	$\Rightarrow$	lo_1	1	$\prec$	fire	1
rtc	=	rt6	:	fire	0	$\Rightarrow$	ro	0	$\prec$	fire	1
rtc	=	rt7	:	fire	0	$\Rightarrow$	ri	0	$\prec$	ro	1
rtc	=	rt8	:	lo	1	$\Rightarrow$	lo_0	1	$\prec$	lo	2
rtc	=	rt9	:	fire	0	$\Rightarrow$	lo_0	1	$\prec$	ri	1
rtc	=	rt10	:	lo	2	$\Rightarrow$	lo_2	2	$\prec$	li	3

CHAPTER 6

RESULTS

This chapter compares the relative timing constraints generated by ARTIST against hand generation in terms of efficiency and quality. The objective of automation is to reduce design time and avoid any error that may be introduced by human interference. It is obvious that one push of a button of ARTIST is much faster than hand generation which would have taken days or even months for a set of designs. The results of verification also indicate that the relative timing constraints generated by ARTIST have the same quality as the hand generation.

6.1 Efficiency

The proof of efficiency and correct functionality of ARTIST demands a large set of example circuits. Recent research on a family of 4-phase latch protocols provides sufficient examples that can be run through the formal verification engine and ARTIST [69].

Birtwistle and Stevens define work that formally and exhaustively investigates all the possible four-phase handshake latch controller protocols in a protocol family. It starts from the most paralleled four-phase latch protocol LC_{max} whose CCS definition is shown in Figure 6.1 and synchronization relationships between L and R channels is shown in Figure 6.2. The Concurrency Workbench (CWB) converts the CCS definition of LC_{max} into a minimized state graph which is shown in Figure 6.3. Then concurrency reduction rules are applied to this minimized state graph which contains 32 states, thus the states are cut away which results in a new protocol with less concurrency than LC_{max}. The exhaustive cut-aways results in 137 related four-phase latch protocols.

The so called **shape** is used to represent cut-away notions. The initial state is denoted as '+', reachable states are denoted as '**o**' and unreachable states are denoted

as '**.**'. Every **shape** represents a distinct 4-phase handshake protocol.

```
R1:   o  o  o  +  o  o  o  o  o
R2:               o  o  o  o  o
R3:               o  o  o  o  o  o  o  o  o
R4:               o  o  o  o  o  o  o  o  o
```

The cut-away is represented by the notation $L_{abcd}\_R_{efgh}$ where L_{abcd} cuts away left side states while R_{efgh} cuts away right side states. L_{abcd} cuts the leftmost a reachable states from R1, the leftmost b reachable states from R2, the leftmost c reachable states from R3 and the leftmost d reachable states from R4 of LC_{max}. R_{efgh}, on the other hand, cuts the rightmost e reachable states from R1, the rightmost f reachable states from R2, the rightmost g reachable states from R3 and the rightmost h reachable states from R4 of LC_{max}. An example of **shape** by cut-away $L_{2112}\_R_{2222}$ is shown below.

```
R1:   .  .  o  +  o  o  o  .  .
R2:               .  o  o  .  .
R3:               .  o  o  o  o  o  o  .  .
R4:               .  .  o  o  o  o  o  .  .
```

The verification of 137 4-phase handshake protocols is performed through the formal verification engine and ARTIST and selected verification results are listed in Table 6.1. The program was run on a workstation configured with Intel® Xeon™ 3.20GHz CPU and 2GB memory. The average number of RT constraints for a protocol generated by ARTIST is 10 and the average ARTIST runtime is 0.15 seconds.

6.2 Quality

The quality of relative timing constraints is measured by the number of constraints that makes the circuit implementation conform to the specification compared to hand generation. The number of relative timing constraints directly determines the working load of pre and postlayout timing validation.

The objective of automating relative timing constraint generation is to largely reduce the design time and maintain the fidelity without any interference of human factors while still retain the quality of the constraints such that the number of RT constraints generated are no more than that of hand-generation.

The set of relative timing constraints generated by hand is listed in the right column of Table 6.2. This set exactly matches the set of {rt200, rt201, rt202, rt203}

in Table 5.5 not only in number but in content. Since the number of timing constraints is the same, one can conclude that there exists a set of relative timing constraints generated by ARTIST such that it has the same quality as the set generated by hand.

However, from the C-Element example described in Section 5.1, there are a total of 25 solution sets of relative timing constraints, 1 of which has 4 constraints, 8 of which have 5 constraints and 16 of which have 6 constraints. The difference in the number of relative timing constraints is caused by the strength of the constraints. Stronger constrains can result in a compact solution set but may over-constrain the design and causes unexpected errors. Weaker constraints are conservative enough to guarantee that they remove the errors while not over-constraining the design. The size of weaker constraints is normally larger than that of stronger constraints.

The hand generation of relative timing constraints are always stronger constraints. Hand design relies on a designer's intuition and familiarity of the circuit structure, plus the experience of asynchronous circuit designer to quickly locate the root cause of the failure. Thus the hand generated constraints by experienced designer are usually optimal constraints.

The set of relative timing constraints generated by ARTIST can be optimized into a smaller size by removing redundant constraints. Remember that there are still 24 solution sets of relative timing constraints for the C-Element that are larger than hand generated set of constraints. By evaluating the strength of relative timing constraints, some weaker constraints can be merged by stronger constraints and result in the same compact size as hand generation.

Here a comparison is demonstrated between an unoptimized set of strict POC constraints {rt190, rt191, rt192, rt193, rt194, rt195} in Table 5.5 and the set of hand generated constraints. The ARTIST generated relative timing constraints with corresponding error traces are shown in the left column of Table 6.2 while hand generated constraints are shown in the right column of the table. The number of hand-generated relative timing constraints is two fewer than this strict POC set of constraints generated by ARTIST.

It can be proven that constraints rt192 and rt194 are redundant and can be covered by constraints rt193 and rt195 by traversing the state transition graph of C-Element

shown in Figure 6.4. By observing the error traces, constraint rt192 is used to remove the failure caused by transition c+ at state 80 while constraint rt193 removes the failure caused by transition bc− at state 70 as well. Although the errors occur at the same level, the failure associated by rt193 is stronger and it removes the subgraph that contains rt192, making rt192 redundant. Notice that all the transitions leaving state 80 are failure transitions. Therefore constraint rt193 is a more appropriate constraint from the perspective of the state graph of the system. Likewise, constraint rt194 can be merged by rt195 in the same way. Now the set of relative timing constraints by ARTIST has 4 constraints {rt190, rt191, rt193, rt195}. But they are still difference from the set of hand generated relative timing constraints. Let up take a close look at constraint rt193 and H3. Constraint H3 removes the whole subgraph down to transition a− at state 50. It is a much stronger constraint compared to rt193. Therefore there may exist multiple sets of relative timing constraints that makes the implementation conform to the specification. The differences between them are just the strength of the constraints employed.

The complete set of relative timing constraints generated by ARTIST must be optimized into a minimized set since redundant constraints increase the working load of pre- and postlayout timing validation. Constraint rt192 and rt194 seem to be suspicious because both of them lead to other failure points as they remove the current failures. When removing the failure directed by c− at state 80, ARTIST is blind and only focuses on the specific error trace returned by the formal verification engine. Hence transitions {b−, bc−} at state 80 and transitions {b−, bc−} at state 70 are all regarded as solution transitions although transition b− at both state 80 and 70 and bc− at state 80 lead to other hazard states. This either results in deadlocks where failure transitions are used as controlling signal transitions by each other such as $c+ \Rightarrow c- \prec bc-$, or uses a stronger relative timing constraint that makes the circuit completely error-free.

The current method for removing redundant relative timing constraints is that after a complete set of RT constraints is generated, each constraint is removed and the formal verification is performed. If verification returns no errors, this temporarily removed constraint is redundant. Otherwise it is a good constraint. This procedure is

not the best way to validate redundant constraints and hence motivates a future work on developing an algorithm that can automatically optimize the set of relative timing constraints to be the minimal one. This work is tied to another investigation on whether choosing different relative timing constraints may have significant difference for timing driven synthesis and place and route since weak and strong aspects of relative timing constraints determines the slack margins of timing.

$$
\begin{array}{rcl}
L & = & lr\uparrow .gS.pV.\overline{la}\uparrow .lr\downarrow .\overline{la}\downarrow .L \\
R & = & gV.\overline{rr}\uparrow .ra\uparrow .pS.\overline{rr}\downarrow .ra\downarrow .R \\
S & = & \overline{gS}.\overline{pS}.S \\
V & = & \overline{pV}.\overline{gV}.V \\
LC_{max} & = & (L|S|V|R)\backslash\{gV, pV, gS, pS\}
\end{array}
$$

Figure 6.1. CCS definition of LC_{max}.

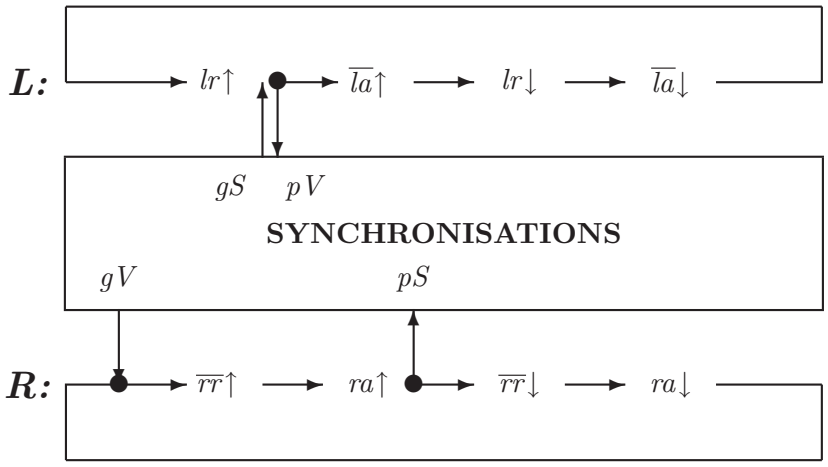


Figure 6.2. Synchronization between L and R channels.

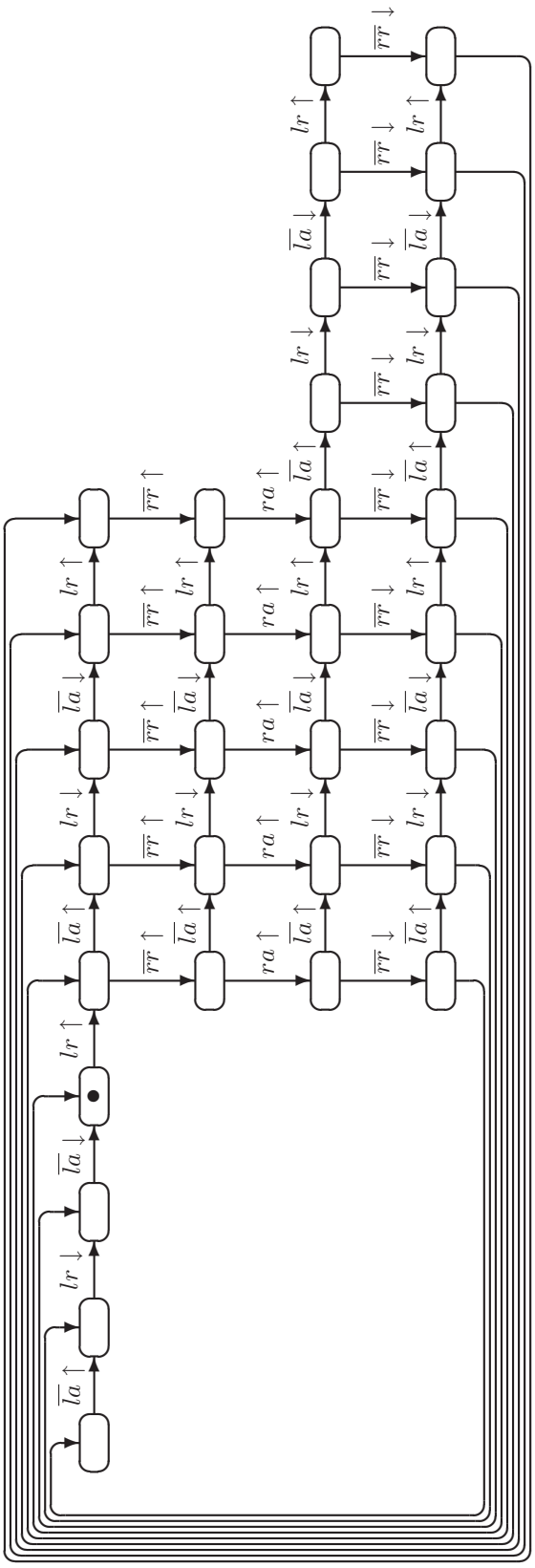


Figure 6.3. State graph of LC_{max}.

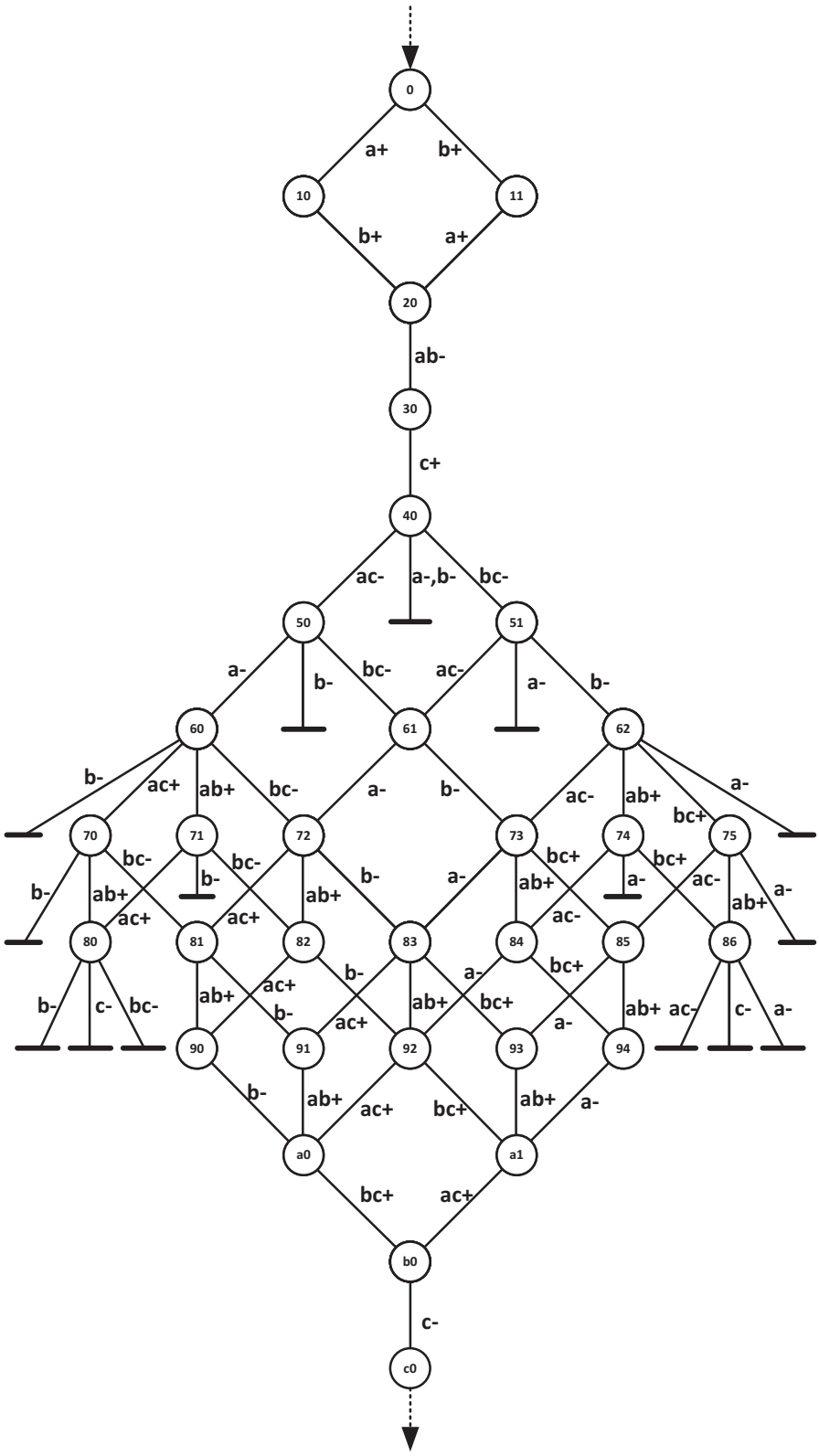


Figure 6.4. State transition graph of C-element.

Table 6.1. Four-phase protocol verification results

No.	Name	#Constraints	Runtime$_{ARTIST}$	Runtime$_{FV}$	#SpecStates	#ImplStates
1	L_{2112}-R_{2222}	9	0.128	0.850	19	113
2	L_{3223}-R_{0020}	2	0.061	0.527	21	104
3	L_{2112}-R_{2022}	7	0.071	56.658	21	395
4	L_{3223}-R_{2044}	16	0.221	1.644	13	95
5	L_{2222}-R_{2242}	26	0.438	1.492	15	124
6	L_{1111}-R_{0044}	12	0.165	1.699	21	335
7	L_{2222}-R_{0020}	10	0.116	0.959	23	145
8	L_{2112}-R_{2264}	2	0.007	0.063	13	26
9	L_{2002}-R_{2262}	3	0.037	0.187	17	49
10	L_{1001}-R_{2262}	13	0.177	4.422	19	114
11	L_{3333}-R_{0042}	16	0.185	1.393	15	136
12	L_{3333}-R_{0020}	24	0.344	2.526	19	177
13	L_{3333}-R_{0000}	29	0.609	4.816	21	326
14	L_{3223}-R_{2042}	15	0.243	1.042	15	143
15	L_{3223}-R_{2022}	9	0.124	0.506	17	106
16	L_{3223}-R_{0042}	16	0.197	2.325	17	210
17	L_{3223}-R_{0022}	14	0.188	1.424	19	150
18	L_{3223}-R_{0000}	12	0.201	3.475	23	275
19	L_{3113}-R_{2242}	4	0.034	0.199	15	52
20	L_{3113}-R_{2222}	4	0.046	0.233	17	70
21	L_{3113}-R_{2042}	8	0.107	0.723	9	158
22	L_{3113}-R_{0040}	6	0.096	4.079	21	261
23	L_{3113}-R_{0022}	6	0.119	2.979	11	318
24	L_{3003}-R_{2042}	19	0.314	13.952	19	390
25	L_{3003}-R_{0022}	15	0.268	17.619	23	352
26	L_{2222}-R_{2022}	10	0.137	1.136	19	106
27	L_{2222}-R_{0040}	9	0.106	0.633	21	131
28	L_{2222}-R_{0022}	6	0.050	0.319	21	80
29	L_{2112}-R_{2042}	12	0.209	1.822	19	344
30	L_{2112}-R_{0042}	22	0.349	15.833	21	1251
31	L_{2112}-R_{0020}	21	0.227	18.869	25	426
32	L_{2002}-R_{2022}	12	0.158	3.119	23	351
33	L_{1111}-R_{2042}	9	0.131	2.993	21	280
34	L_{1111}-R_{0022}	4	0.060	0.583	25	136
35	L_{1001}-R_{2042}	4	0.048	0.452	23	291
36	L_{3333}-R_{0044}	2	0.015	0.138	13	52
37	L_{3113}-R_{2044}	3	0.028	0.289	15	68
38	L_{3113}-R_{0044}	1	0.010	0.112	17	65
39	L_{2002}-R_{2222}	4	0.070	0.464	21	85
40	L_{2222}-R_{2222}	5	0.032	0.223	17	106
41	L_{3113}-R_{2022}	10	0.126	1.667	19	220
42	L_{3113}-R_{0042}	7	0.100	4.465	19	272
43	L_{0000}-R_{2242}	25	0.931	44.525	23	1152
44	L_{0000}-R_{2244}	6	0.088	0.454	21	125
45	L_{0000}-R_{2262}	12	0.197	2.359	21	340
46	L_{0000}-R_{4044}	4	0.049	7.069	21	515
47	L_{0000}-R_{4264}	18	0.226	1.061	17	173
48	L_{1001}-R_{2242}	6	0.090	0.851	21	203
49	L_{1001}-R_{2244}	12	0.210	1.363	19	200
50	L_{1001}-R_{4264}	7	0.043	0.378	15	127
51	L_{1111}-R_{2044}	11	0.090	0.773	19	130
52	L_{1111}-R_{2222}	7	0.111	0.644	21	135
53	L_{1111}-R_{2242}	4	0.042	0.341	19	91
54	L_{1111}-R_{2262}	5	0.048	0.427	17	79
55	L_{1111}-R_{2264}	4	0.057	0.289	15	56
56	L_{2002}-R_{2244}	4	0.036	0.171	9	49
57	L_{2002}-R_{2264}	4	0.038	0.168	15	45
58	L_{2002}-R_{4244}	4	0.042	0.171	15	50
59	L_{2112}-R_{2244}	4	0.034	0.173	15	52
60	L_{2112}-R_{2262}	16	0.216	2.301	15	137
	Average	10	0.150	3.930	18	207

Table 6.2. Unoptimized RT constraints and corresponding traces versus hand-generated constraints for C-Element.

ARTIST Generated			Hand Generated	
rt190	`a b ab c a`	c+ $\mapsto$ ac- $\prec$ a-	H1	c+ $\mapsto$ ac- $\prec$ a-
rt191	`a b ab c b`	c+ $\mapsto$ bc- $\prec$ b-	H2	c+ $\mapsto$ bc- $\prec$ b-
rt192	`a b ab c ac a ac ab c`	c+ $\mapsto$ bc- $\prec$ c-	H3	c+ $\mapsto$ bc- $\prec$ a-
rt193	`a b ab c ac a ac ab bc`	c+ $\mapsto$ bc- $\prec$ ab+	H4	c+ $\mapsto$ ac- $\prec$ b-
rt194	`a b ab c bc b bc ab c`	c+ $\mapsto$ ac- $\prec$ c-		
rt195	`a b ab c bc b bc ab ac`	c+ $\mapsto$ ac- $\prec$ ab+		

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Asynchronous circuits have power and performance benefits over its synchronous counterpart. However asynchronous design is not widely adopted in industry due to a lack of CAD tools, and requiring deep expertise in asynchronous circuit design. A relative timing based asynchronous design methodology allows synchronous design engineers to design asynchronous circuits using conventional clocked CAD tools with little asynchronous circuit knowledge by making use of precharacterized asynchronous templates.

The core of this design methodology is the asynchronous template characterization that employs formal model checking and a set of relative timing constraints. These were previously manually generated such that the circuit implementation conforms to the specification. This manual generation of relative timing constraints is very time consuming and prone to errors. It may take hours or even days for an experienced design engineer to generate a complete set of relative timing constraints that guarantee the correctness of design.

This thesis presents algorithms for automatically generating relative timing constraints with the aid of a bisimulation semantic based formal verification engine. Path based relative timing constraints restrict the relative delays between two paths from a common point of divergence to the point of convergence by incorporating different relative arrival times of the two racing events. These algorithms remove any possibility of internal glitches and nonconformance between the implementation and the specification.

The algorithms are implemented in the tool ARTIST as an embedded function call of the formal verification engine *Analyze*. The fundamental principle for resolving errors is to enforce other concurrent transitions to occur before a failure transition

such that the failure states are made unreachable. All the necessary information required for building a trace status tableau is collected from *Analyze*. The generation of relative ordering and the common point of divergence is created by searching and backtracking the trace status tableau.

The set of relative timing constraints generated by ARTIST is compared against hand generated constraints in terms of efficiency and quality. It is obvious that one push of the button of ARTIST is much more efficient than hand generation. The verification result on over 100 4-phase latch controllers through concurrency reduction shows the average number of relative timing constraints for a protocol generated by ARTIST is 10 and the average runtime is 0.15 seconds per design. The quality of relative timing constraints refer to the number of relative timing constraints because the number of constraints are directly related to the working load of postlayout timing validation. Since ARTIST generates weaker constraints, the solution sets of relative timing constraints are equal or more than hand generated constraints. Those sets that have more constraints may be optimized into a smaller size equal to the hand generation. Therefore the sets of relative timing constraints generated by ARTIST is much more efficient while retaining the same quality as hand generation.

The algorithms also support user-specified input signals as the point of divergence of relative timing constraint such that it can be mapped to the reference virtual clock pin to facilitate pre- and postlayout timing validation.

7.2 Future Work

The algorithms described in this thesis generate all the possible sets of relative timing constraints that can make the implementation conform to the specification. Some of them are composed of more constraints which are relatively weak and some of them are composed of fewer constraints which are relatively strong. The most compact set of relative timing constraints can be generated by specifying the breadth first option, but this takes more time and consumes more memory since every constraint node at each level must be evaluated by the formal verification engine and can then move to next level. A designer may choose the depth first option to return a quick set of relative timing constraints which may not be the most compact set. To release

the burden of pre- and postlayout timing validation, it is imperative to have some algorithm that can optimize such a noncompact set of relative timing constraints into an minimized one by removing redundant constraints. This algorithm may be implemented by observing the traces of solution relative timing constraints. One relative timing constraint can be regarded as redundant if the state node it applied to has already been unreachable by enforcing other constraints.

Other future work is to investigate the impact of the different margins of relative timing constraints on timing driven synthesis and place and route in terms of area, power and performance. A single error can be resolved by multiple candidate relative timing constraints. The weak and strong aspects of a relative timing constraint determines the relative timing margin. The impact of choosing different margins of relative timing constraints on the design has not been explored. If a loose margin and an aggressive margin do have a difference in area, performance and power, the relative timing constraints may be carefully chosen and traded off for timing driven synthesis and place and route to gain the optimal results. The current algorithms generate relative timing constraints to be specific to a single error trace returned from formal verification engine. ARTIST only focuses on resolving current failure instead of considering other failures a controlling event may potential lead to. This results in many redundant relative timing constraints. Therefore choosing a proper relative timing constraint becomes important in achieving an optimal design.

Although the incompatibility of single track of GasP family asynchronous circuits with formal verification engine is resolved by re-partitioning the pipelined GasP into a double track structure, the branch and merge modules of GasP family remain unexplored. The major difficulty is the inability of analyzing the non-determinism of GasP merge module with respect to tracking causalities. An alternative modeling may be needed. Once the individual GasP modules are thoroughly verified, a system level integration for large GasP application can be performed.

REFERENCES

[1] J. You, Y. Xu, H. Han, and K. S. Stevens. "Performance Evaluation of Elastic GALS Interfaces and Network Fabric." In *Elsevier Electronic Notes in Theoretical Computer Science*, Vol. 200, No. 1, pages 17-32, February 2008.

[2] D. E. Muller. "Asynchronous logics and application to information processing. " In H. Aiken and W. F. Main, editors, *Proc. Symp. on Application of Switching Theory in Space Technology*, pages 289-297. Stanford University Press, 1963.

[3] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching.* Harvard University Press, Apr. 1959, pp. 204–243.

[4] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design – A Systems Perspective.* Kluwer Academic Publishers, 2001.

[5] K. S. Stevens, P. Golani, and P. A. Beerel. "Energy and Performance Models for Synchronous and Asynchronous Communication." In *IEEE Transactions on Very Large Scale Integration*, 2010.

[6] K. S. Stevens. "Energy and Performance Models for Clocked and Asynchronous Communication." In *9th International Symposium on Asynchronous Circuits and Systems*, May 2003, pp. 56-66.

[7] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors,* 2005 edition.
http://www.itrs.net/links/2005itrs/design2005.pdf

[8] L. S. Nielsen. "Low-power Asynchronous VLSI Design." PhD thesis, Department of Information Technology, Technical University of Denmark, 1997.

[9] A. J. Martin. "The Limitations to Delay-Insensitivity in Asynchronous Circuits ", *Sixth MIT Conference on Advanced Research in VLSI*, 1990.

[10] K. S. Stevens, D. Gebhardt, J. You, Y. Xu, V. Vij, S. Das, and K. Desai. " The Future of Formal Methods and GALS Design." In *Electronic Notes in Theoretical Computer Science*, Vol. 245, No.1, pages 115-134, August 2009.

[11] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamjura, "TITAC: Design of a quasi-delay-insensitive microprocessor," *IEEE Design Test Comput.*, vol. 11, pp.50-63, Feb. 1994.

[12] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC-2: An asynchronous 32-bit microprocessor based on scalable delay insensitive model," in *Proc. ICCD'97*, pp.288-294.

[13] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings and T. K. Lee, "The Design of an Asynchronous MIPS R3000 Microprocessor", *Proc. 17th Conference on Advanced Research in VLSI*, 164-181, IEEE Computer Society Press, 1997.

[14] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple, "SPA - A synthesisable amulet core for smartcard applications," in *Proc. International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp.201-210.

[15] T. Verhoeff, "Delay-insensitive codes: An overview," *Distrib. Comput.* 3 (1988), pp. 1-8.

[16] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct. 1992, pp. 279–282.

[17] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," in *IEEE Transactions on VLSI Systems*, 1(2), June, 1993.

[18] C. J. Myers, "Computer-aided synthesis and verification of gate-level timed circuits," Ph.D. dissertation, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[19] C. J. Myers, Asynchronous Circuit Design John Wiley & Sons, July 2001.

[20] I. Sutherland and S. Fairbanks, "GasP: A Minimalist FIFO Control," *Proc. of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2001.

[21] I. E. Sutherland, R. F. Sproull, and D. F. Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.

[22] I. E. Sutherland and J. K. Lexau, "Designing fast asynchronous circuits," in *7th International Symposium on Asynchronous Circuits and Systems*, Mar. 2001, pp. 184–193.

[23] E. M. Clarke, O. Grumberg and D. A. Peled. Model Checking. MIT Press, 1999.

[24] R. E. Bryant. "Graph-based algorithms for boolean function manipulation." *IEEE Transactions on Computers*, 1986.

[25] M. Fujita, H. Fujisawa, and N. Kawato. "Evaluation and improvement of boolean comparison method based on binary decision diagrams." In *Proceedings of IEEE International Conference on Computer Aided Design*, IEEE Computer Society Press, 1988.

[26] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincenteli. "Logic verification using binary decision diagrams in a logic synthesis environment." In *International Conference on Computer-Aided Design*, pp. 6-9, 1988.

[27] Accellera. PSL Reference Manual. http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[28] B. Cohen, S. Venkataramanan, and A. Kumari. *SystemVerilog Assertions Handbook* VhdlCohen Publishing, 1st edition, 2005.

[29] D. L. Perry and H. D. Foster. *Applied Formal Verification* Electronic Engineering, McGraw-Hill, 2005.

[30] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative Timing," in *Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 208–218, April 1999.

[31] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative Timing," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1), Feb. 2003, pp. 129–140.

[32] K. S. Stevens, Y. Xu, and V. Vij, "Characterization of Asynchronous Templates for Integration into Clocked CAD Flows," *15th International Symposium on Asynchronous Circuits and Systems*, pp. 151-161, May 2009.

[33] N. Andrikos, L. Lavagono, D. Pandini, and C. P. Sotiriou, "A Fully-Automated Desynchronization Flow for Synchronous Circuits," In *Design Automation Conference*, pages 982-985. ACM/IEEE, June 2007.

[34] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, " Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1904-1921, Oct 2006.

[35] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake protocols for de-synchronization. In *International Symposium on Asynchronous Circuits and Systems*, pages 149-158. IEEE, Apr 2004.

[36] K. Y. Yun and D. L. Dill, "Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementation)," *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 2, pp. 101–117, Feb. 1999.

[37] S. M. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," Ph.D. dissertation, Stanford University, Department of Computer Science, 1993.

[38] Robin Milner. *Communication and Concurrency.* Computer Science. Prentice Hall International, London, 1989.

[39] P. Stevens, "Concurrency Work Bench," http://homepages.inf.ed.ac.uk/perdita/cwb/.

[40] E. Quist, P. Beerel, and K. S. Stevens, "Enhanced SDC Support for Relative Timing Designs," In *Digital Automation Conference*, User Track Poster, July 2009.

[41] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivettie, M. Pistore, M. Roveri, and A. Tchaltsev, "Nusmv 2.4 user manual". http://nusmv.irst.itc.it.

[42] K. Desai, and K. S. Stevens, "Scalable Asynchronous Hardware Protocol Verification for Compositions with Relative Timing," In the *TAU 2010 Worshop*, March, 2010.

[43] K. S. Stevens, "Practical Verification and Synthesis of Low Latency Asynchronous Systems," Ph.D. dissertation, University of Calgary, Calgary, Alberta, Canada, September 1994.

[44] C. A. R. Hoare, *Communicating Sequential Processes.* London: Prentice Hall International, 1985.

[45] ——, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, August 1978.

[46] J. Peterson, *Petri Net Theory and Modeling of Systems.* Prentice Hall, 1981.

[47] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli, "A unified signal transition graph model for asynchronous control circuit synthesis," in *International Conference on Computer-Aided Design(ICCD).* IEEE Computer Society Press, Nov. 1992, pp. 104-111.

[48] D. L. Dill, "Trace theory for automatic hierarchical verification of speed-independent circuits," *An ACM Distinguished Dissertation*, MIT Press, 1989.

[49] D. L. Dill, S. M. Nowick, and R. F. Sproull, "Specification and automatic verification of self-timed queues." *Formal Methods in System Design*, vol. 1, no. 1, July 1992.

[50] S. M. Nowick and D. L. Dill, "Practicality of state-machine verification of speed-independent circuits," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD),* Nov. 1989, pp. 266-269.

[51] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. M. Nowick, "A corretness criterion for asynchronous circuit validation and optimization," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* vol.13, no. 11, Nov 1994.

[52] R. Paige and R. Tarjan, "Three partition refinement algorithms," *SIAM Journal of Computation*, vol, 16, no. 6, pp.973-989, 1987.

[53] J, -C. Fernandez, "An implementation of an efficient algorithm for bisimulation equivalence," *Science of Computer Programming*, vol. 13, pp. 219-236, 1990.

[54] J. -C. Fernandez, ""On the fly" Verification of Behavioral Equivalences and Preorders," in *Prceedings of CAV'91,* ser. LNCS, K. G, Larsen and A. Skou, Eds., no. 575, 1991, pp.181-191.

[55] H. Kim, P. A. Beerel, and K. S. Stevens, "Relative timing based verification of timed circuits and systems," in *8th International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 115–126.

[56] T. Yoneda, T. Kitai, and C. Myers, "Automatic derivation of timing constraints by failure analysis," in *Computer Aided Verification (CAV'02)*, pages 195-208, July 2002.

[57] T. Kitai, T. Yoneda, and C. J. Myers, "Failure trace analysis of timed circuits for automatic timing constraints derivation", in *IEICE Transactions on Inf. and Syst.,* vol. E88-D, no. 11, Nov 2005.

[58] T. Yoneda and H. Ryu. "Timed trace theoretic verification using partial order reduction. *Proc. of Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems,* pages 108-121, 1999.

[59] T. Yoneda. "VINAS-P: A tool for trace theoretic verification of timed asynchronous circuits. *LNCS 1855 Computer Aided Verification*, pages 572-575, 2000.

[60] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. "Protocol verification as a hardware design aid". In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522-525, Cambridge, MA, October 1992. IEEE Computer Society.

[61] Y. Xu, and K. S. Stevens. "Automatic Synthesis of Computation Interference Constraints for Relative Timing Verification." In *26th International Conference on Computer Design*, pp. 16-22, October, 2009.

[62] W. S. Coates, J. K. Lexau, I. W. Jones, S. M. Fairbanks, and I. Sutherland, "FLEETzero: An Asynchronous Switching Experiment," *Proc. of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2001.

[63] K. van Berkel and A. Bink, "Single-Track Handshaking Signaling with Application to Micropipelines and Handshake Circiuts," *Proc. of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996.

[64] M. Ferretti and P. Beerel. "High Performance Asynchronous Design Using Single-Track Full-Buffer Standard Cells." *IEEE Journal of Solid-State Circuits*, 41(6):1444-1454, 2006

[65] M. Nystrőm, E. Ou, and A. Martin. "An Eight-Bit Divider Implemented with Asynchronous Pulse Logic." In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems*, pages 229-239, 2004.

[66] I. Sutherland, "A Six Four GasP Tutorial." Technical Report, UCIES2007-is49 at http://research.cs.berkeley.edu/class/fleet/docs/, 2007.

[67] P. Joshi. "Static Timing Analysis of Gasp." Master of Science Thesis, Electrical Engineering, Faculty of the USC Viterbi School of Engineering, University of Southern California, Dec. 2008.

[68] S. M. Gilla, M. Roncken, and I. Sutherland. "Long-Range GasP with Charge Relaxation". In *Proceeding Sixteenth IEEE International Symposium on Asynchronous Circuits and Systems*, 2010

[69] G. Birtwistle and K. S. Stevens, "The family of 4-phase latch protocols," in *14th International Symposium on Asynchronous Circuits and Systems*. IEEE, April 2008, pp. 71–82.