Specification, Simulation, and Verification of Timing Behavior

by

Tod Amon

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

Doctor of Philosophy

University of Washington

1993

Approved by____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree______________________________________

Date____

University of Washington

Abstract

Specification, Simulation, and Verification of Timing Behavior

by Tod Amon

Chairperson of the Supervisory Committee: Professor Gaetano Borriello
Department of Computer Science
and Engineering

Temporal behavior needs to be formally specified, validated, and verified, if systems that interface with the outside world are to be synthesized from high-level specifications. Due to the high level of abstraction, the work presented in this thesis applies to systems that ultimately can be implemented using hardware, software, or a combination of both.

In the area of specification, a generalization of the event-graph specification paradigm is presented. The model supports the expression of complex functionalty using an operational semantics and cleanly integrates structure into the event-based paradigm. Temporal relationships between systems events are specified using a denotational semantics that relies on both chronological and causal relationships to identify the discrete event occurrences being constrained.

It is of critical importance that a design representation support user validation. A simulator, called OEsim, has been implemented to support validation. The simulator can report timing constraint violations, and detect inconsistencies between the operational and denotational specifications.

Synthesis algorithms can exploit guarantees regarding temporal behavior to produce efficient implementations, and often need timing information in order to properly synthesize designs from abstract specifications. Two verification tools are presented in this dissertation. The first demonstrates the feasibility and benefits of performing symbolic timing verification, in which variables are used in place of numbers. The second addresses a fundamental problem: determining how synchronization affects the temporal behavior of concurrent systems.

Table of Contents

List of Figures

Acknowledgments

I would first like to thank my wonderful advisor, Gaetano Borriello, who offered support and guidance for which I am most grateful. He helped develop and encourage my interest in this area of research, and he gave me a great deal of his time, an invaluable contribution. I have benefited greatly from his ability to see the bigger picture, and his insights about conducting research.

I sincerely appreciate the efforts of the other members of my supervisory committee, Steve Burns, Ted Klastorin, and Alan Shaw, for the time and effort they have spent reading my work and helping me with this dissertation. Three good friends, Pai Chou, Henrik Hulgaard, and Ross Ortega, read the very rough drafts and deserve utmost praise for their patience and fortitude.

Two years ago I contemplated trying to finish in one year. I am so glad that I did not. Working closely with Steve and Henrik on the material of Chapter 7 has been very rewarding. I learned so much, and had so much fun doing so. Both deserve many thanks for so very many things.

I have been fortunate enough to receive assistance from two technical staff members, Wayne Winder, and Karen Bartlett, who wrote some of the code and more importantly helped me with the endless stream of mysterious bugs that I was so adept at generating.

The Department of Computer Science at the University of Washington is a most friendly place. As a result, there are many other people, past and present, that shared this experience with me, whom I will remember with great fondness, for they made my time here most enjoyable and memorable.

Finally, I wish to express special thanks and gratitude to a dear friend and companion, my wife Karen. She was with me at the very beginning, inspiring me to turn my dreams and goals into this reality. She and I are heading in different directions soon, but her love and support helped me make it through, and she more than anyone is deserving of my thanks.

To my family, especially Carey.

Chapter 1

Introduction

Design automation can be loosely divided into four major areas:

- *Specification.* Systems must be described and specified formally if they are to be analyzed or manipulated by other design tools.
- *Validation.* Specifications need to be natural, formal, and correct—they should capture the system envisioned by their human designer ("*Did I specify what I wanted?*"). Simulation, i.e., execution of the specification, helps users validate a specification with respect to both functionality and performance.
- *Synthesis.* Designs can be synthesized and transformed from abstract specifications into physical implementations.
- *Verification.* Verification tools formally analyze designs and requirements and provide assurance that designs have been properly implemented and that they will always work correctly.

The growing complexity of digital circuits and other real-world systems requires that design automation tools work at increasingly higher levels of abstraction. Abstraction helps manage complexity because systems can be specified by their behavior instead of their structure. Behavior is a high-level description of *what* a system should do, whereas structure is a low-level description of *how* a system will be implemented. For example, a digital circuit can be described as an algorithm (a behavioral perspective) or as a set of interconnected transistors (a structural perspective). There are, of course, many different levels of abstraction and there exists a large body of software to help automate the design process at each level.

Most computer-aided design tools focus on a system's functional characteristics. Functional specifications emphasize data transformations and the sequence of operations that are performed when system inputs are translated into system outputs. Temporal characteristics are often considered by design tools that work with specific low-level abstractions (e.g., in synchronous digital circuits, synthesis algorithms concentrate on setup and hold times). At higher levels of abstraction, however, design tools have often ignored system timing.

	Behavior	*Structure*
Functional	abstract concurrent program (not necessarily implementable)	register-transfer level descriptions
Timing	real-time constraints (context dependent)	propagation delays setup/hold times

Figure 1.1: The design representation space is divided between behavior and structure and, orthogonally, between functional and timing aspects.

There are many reasons for this omission. At higher levels of abstraction, systems should be specified without concern for the specific timing discipline which will be used in the system implementation. Performance may be important (e.g., optimize for delay) but a common belief has been that timing issues do not affect system correctness. This assumption has been especially valid for the systems for which high-level digital circuit synthesis techniques have been developed (e.g., processor and data path intensive designs). These designs are often based on a simple synchronous timing model and are not constrained by complex timing issues.

Timing issues have certainly been a focus of attention in the real-time systems community. Many of the the problems addressed by the real-time community are outside even the general scope of this dissertation, because probabilistic information is used to analyze stochastic behavior, or because the problems involve complex real-time systems running on sophisticated processors (e.g., for airline reservations, flight control, etc.). However, since process control is becoming increasingly more distributed, smaller embedded systems are receiving more attention. New technologies are blurring the traditional line between software and hardware, and reactive *hard* real-time systems are becoming important to both the real-time systems and the design automation communities.

1.1 Importance of Timing Behavior

Why has the specification and analysis of timing behavior (the subject of this dissertation) become such an important area of research? One contributing factor is the realization that many aspects of a system's behavior are not under the control of the designer. Systems must conform to the environment in which they will be placed. The environment may demand that particular timing relationships be respected. If synthesized designs need to be integrated into pre-exising systems, then synthesis tools must accommodate the environment and the timing relationships that need to be satisfied.

Since design automation tools are working with higher levels of design abstraction, timing issues are becoming especially important. When a design is specified at a high level, many important trade-offs are made during synthesis. There are many possible implementations and it may be that part of the design can be in hardware and part in software. One important criterion that can be used in making decisions is an analysis of the resulting timing behavior of a system and whether or not it will meet the timing constraints provided by the designer. For example, given information about input rates, a synthesis algorithm could appropriately choose between a fast expensive high performance component or an inexpensive and slower alternative.

Formal verification is becoming an area of renewed interest as designs are becoming increasingly more complicated. Verification tools are used to formally reason about system correctness, and reasoning about complex system behavior often involves reasoning about time. The complexity of current systems has also led to interest in executable specification languages for performing design validation. If a specification contains timing information, then users need to validate the timing behavior of their specification as well as its functionality.

Design automation tools need to take timing behavior into account. This is by no means an easy task, because time adds a new dimension of complexity to problems that in many cases are already quite difficult. It cannot be ignored, however, because many of the problems that need to be automated are constrained or driven by timing concerns.

1.2 Contributions

All four areas of design automation (specification, validation, synthesis, and verification) are discussed in this dissertation. However, synthesis methodologies and algorithms will not be presented, even though much of this work has been motivated by the goal of being able to synthesize designs specified at high levels of abstraction. We need to be able to specify timing behavior in order to achieve this goal, and synthesis algorithms need to use formal analysis techniques that can determine (verify) the timing behavior of specifications. This analysis provides information which can be used to guide the synthesis process and establish the formal correctness of synthesis transformations. Thus, this dissertation does not address the synthesis problem directly, but tackles many of its prerequisites.

Some areas of design automation are well developed, and current research is focused on developing incremental improvements to existing techniques. In contrast, the inclusion of timing issues into design automation is still in its infancy. For example, the characteristics of a good specification language for timing behavior are still being identified. Thus, the concepts and philosophies espoused in the area of specification are as important as the syntax, semantics, and design tools that are presented.

In the area of verification, the contributions are more concrete. A fundamental problem in timing verification is solved and we present a complete theory, with proofs and efficient algorithms. This work is tightly focused on analyzing the timing behavior of systems that are specified as synchronizing concurrent processes. The results are general and in fact have very little direct relationship to digital circuits. This is because the abstraction level is quite high, and the system being described could be implemented as hardware, software, or a combination of both.

The major contributions are briefly described in the following two subsections, which summarize the content of this dissertation as a whole. Various aspects of this work have been presented to the research community and reported in the literature: [Amon et al. 91], [Amon & Borriello 91a], [Amon & Borriello 91b], [Amon & Borriello 92], [Amon et al. 93].

1.2.1 Specification and Validation

The primary contribution is the presentation of a formal and general representation for timing specification based on a bipartite graph model augmented with a restricted first-order predicate calculus to specify timing relationships between signal events. It can be viewed as a natural extension of event-based representations, with support for higher levels of functional abstraction, and the recognition that timing constraints are relationships between occurrences of circuit events, and should not be represented simply as edges in event graphs.

A simulator has been implemented for the representation and provides validation capabilities by incrementally checking timing constraints for violations as the simulation progresses. The representation is suitable for rapid prototyping because circuits can be represented by either an abstract specification of their interface behavior or by a detailed description of their internal logic structure (or by a mixture of both). Experimentation with the simulator permits the user to study the interaction of complex timing behaviors. With respect to synthesis and verification, this new representation serves as an exploratory framework to help identify the problems that timing issues introduce without *a priori* restricting the classes of circuits or behaviors that can be described.

1.2.2 Timing Verification

The primary contribution is the presentation of non-stochastic verification techniques to determine minimum and maximum separation times between constrained system events (in any possible execution). The timing behavior of the system is specified by providing lower and upper bounds on delays between causally related events. Timing analysis is required in order to determine how concurrency, communication, and synchronization affect the system's temporal behavior. Two very different verification methodologies are presented.

Symbolic timing verification is a powerful extension to traditional constraint checking in which delays and constraints can be expressed using variables instead of numbers. The techniques are quite powerful because they yield not only simple bounds on delays (e.g., set $X > 20$ if the system is to work) but also relationships between variables, which expose design trade-offs (e.g., if X is small then Y can be large). Our symbolic timing verifier uses symbolic linear programming and a rule based verification strategy. The verifier can handle non-deterministic conditional behavior and multiple processes that communicate in a single direction. Rules have been developed to support pessimistic verification (in that weak assumptions can be used in the verification process) of two-way communication, however verification is rarely successful because the verification rules do not adequately address this complicated problem.

Our second verifier was designed specifically to solve the problem of verifying communicating concurrent processes. Communicating processes are modeled as a cyclic connected graph. The nodes of the graph represent events and the arcs dependencies (propagation delays) between these events. This model permits the representation of both blocking and non-blocking communication. In the blocking case, two interacting processes wait for a pair of synchronizing events (one in each process) to occur and then both proceed past the synchronization point. In the non-blocking case, only the receiving process waits while the sender can proceed. The blocking model is popular in the software community as well as in self-timed circuit synthesis. The non-blocking model is popular in the hardware community and is used in interface and asynchronous circuit specification.

Depending on the level of abstraction in the specification, events may represent low-level signal transitions at a circuit interface or control flow in a more abstract behavioral view. We describe algorithms that determine exact bounds on the separation in time between system events. We present a theoretical framework for solving this problem for strongly connected process graphs without conditional or iterative behavior, and develop an efficient algorithm based on this theoretical foundation.

1.3 Top-Down vs. Bottom-Up

There exists a natural tension between the expressivity of specifications, and the amount of automation which can be provided. Simple specifications are easier to analyze than complicated ones, but simple specifications may not be expressive enough to specify systems (and allow analysis) that designers want automated. Research that extends design automation's ability to take timing behavior into account can approach the problem from one of two directions:

- From the *top-down*, emphasizing expressivity and solving problems that designers face (of course the complexity of this problem means that the solutions may not be optimal), or,
- From the *bottom-up*, emphasizing exact and efficient solutions to fundamental problems that may serve as a foundation upon which practical design tools can be built.

Neither approach is inherently better than the other. One advantage of bottom-up is that, over time, exact solutions to complex problems can be developed. However, because there are many different directions that can be taken, the research may be purely theoretical and not lead towards the solution of real-world problems. Top-down research is more closely aligned with *current* needs, but often must rely on heurestics and other non-optimal strategies to solve complex problems. Top-down research, however, can provide a target for the bottom-up approach.

1.4 Dissertation Overview

This dissertation is divided into two major parts. The next three chapters take a top-down approach, and addresses the specification of complex timing behavior. Chapter 2 presents related work in this area and provides an in-depth examination of the event-based representation paradigm and the features that a good representation for timing behavior should possess. Chapter 3 introduces a new representation, *operation-event graphs* (OEgraphs), which have many of these features. Chapter 4 presents OEsim, a simulator which can be used to validate designs specified as OEgraphs. The simulator performs constraint checking during simulation and is an example of the benefits of the top-down approach, in that it can be used for rapid prototyping.

Simulation can be used to analyze behavior, but only for one specific set of inputs and propagation delays. More powerful techniques are needed in order to establish bounds on all possible system behaviors. Chapters 5 through 7 are based on a more bottom-up approach, and present work in analyzing the timing behavior of a restricted class of concurrent processes. Chapter 5 presents related work in this area and provides an overview of the verification/analysis task. OEgraphs are capable of expressing complex timing behavior but, due to this complexity (as discussed in Chapter 3), verification is not generally feasible. In Chapter 6 we present novel work in symbolic timing verification for a restricted form of OEgraphs. Verification can be performed even if some of the propagation delays remain unspecified (as symbolic variables). Chapter 7 considers a very restrictive set of timing behavior for which a precise formal semantics, analytical techniques, proofs, and algorithms are provided. The analysis computes the time separation of events in communicating concurrent processes. It is an example of the benefits of the bottom-up approach, in that the solution to this problem may serve as a foundation upon which other design tools can be built.

In Chapter 8 we conclude and discuss some limitations to our specification and verification methodologies. Future directions of possible research are also discussed.

Chapter 2

Specification

Specification languages are, in many ways, the cornerstones upon which design tools are built. They constitute a medium through which a human designer and a design tool communicate. A specification language provides a notation (a syntactic domain) and an interpretation (a semantic domain) which are used by a designer to express the characteristics, requirements, and properties of an envisioned design.

Specification languages have a domain of applicability—some are tightly focused on specific systems while others are more general. For example, a specification language used for telecommunication circuits implemented using commercial DSP chips would have a small domain of applicability. A general purpose language might support more abstract descriptions of both parallel and sequential activity and not assume any particular implementation strategy.

In the next three chapters, we loosely restrict our domain to specification languages for digital circuits with the primary emphasis being the specification of their timing behavior. Because we are concerned with specifications that are potentially quite abstract, e.g., specifications that do not define the circuit structure, much of this work may also relevant to other domains like software. In later chapters, we will analyze the timing behavior of systems specified at very high levels of abstraction and the focus on digital circuits will be even further diminished.

Most specification languages restrict the class of digital circuits (or behaviors) that can be described. This is perfectly reasonable when the emphasis is on the accompanying design tools and their use in design automation. Our emphasis is on a general specification language for timing behavior as opposed to the creation of a specific design tool. We are interested in a language that can serve as an exploratory framework to help expose and examine the complex issues that arise when design automation attempts to take timing behavior into account.

The remainder of this chapter is divided up into three sections. First, we examine the basic elements and properties that we believe should be present in a general specification language for timing behavior. Next, we review related work, and discuss some of the strengths and weaknesses of existing representations and specification paradigms. The last section is devoted to one specific paradigm which we believe is particularly well equipped to meet our objectives.

2.1 Important Properties of Specification Languages

We need to identify what qualities we want in a specification language if we are going to be able to evaluate existing representations, and provide sufficient justification for the creation of yet another new specification language. The overall goal, of course, is to support the development of design automation software that can deal with timing behavior. We are interested in specification languages that support synthesis, validation, and verification. In this section, we present a general set of specification language requirements: formality, expressiveness, abstraction, and support for validation.

2.1.1 Formality

Of all the properties that a specification language should have, *formality* is undoubtedly the most important. A specification needs to have a sound mathematical basis since it is a piece of communication akin to a contract between a human designer and the formal methods which operate upon the design.

Informal specifications are by their very nature ambiguous. One of the primary benefits of formalizing a specification is that any ambiguity can be resolved. Specification languages thus need to be structured so that every "well formed" specification is inherently unambiguous.

There are many other benefits of formal specification. We refer readers to [Meyer 85] for a more detailed discussion. We also refer readers to [Wing 90] for a summary of formal methods and formal design specification.

2.1.2 Expressiveness

Specification languages are designed for a particular domain of applicability, and need to have enough expressive power to represent designs from that domain. Our domain is a very large one: digital circuits, and we are specifically interested in the specification of timing relationships.

Timing relationships must be specified at many different levels of abstraction. Although our primary interest is the specification of timing constraints at high levels of abstraction, lower-level timing relationships are also of interest. This is because many different levels of abstraction may be present in a single specification. At all levels of abstraction, many timing relationships are conceptually quite similar. For example, outputs are delayed with respect to inputs, and propagation delays need to be specified. Relationships can often be expressed by specifying minimum and/or maximum separation times between system "events." For example, at low levels of abstraction, setup and hold times can be specified. At higher levels of abstraction, required sampling rates and response time latencies can be defined.

Some timing constraints are tightly coupled with the abstraction level. For example, various clocking methodologies should be specifiable. This includes designs that are asynchronous, synchronous (single or multi-phase), edge-triggered, level-sensitive, etc. At higher levels of abstraction, timing constraints are often context dependent. For example, different constraints might apply depending upon the priority of a request or the current "mode" of operation.

We are particularly interested in being able to express timing relationships at higher levels of abstraction. As mentioned in the introduction, higher levels of abstraction are needed in order to manage the growing complexity of digital circuits. It should be possible to specify the behavior of a design without imposing an implementation bias. The design tools that operate upon the specification can then make use of the available freedom to produce an efficient implementation. The ability to specify timing relationships is of utmost importance if this freedom is to be exploited. Re-use of existing designs is, of course, another important way to help manage complexity, and specification languages need to support multiple levels of abstraction.

We are interested in specifying and analyzing non-stochastic temporal behavior. With respect to circuit structure, delays are often specified by providing minimum, typical, and maximum values. This is presumably a safe abstraction for a very complex problem (actual component delays are dependent upon many factors including resistance, capacitance, fabrication parameters, temperature, etc.). At higher levels of abstraction, response time constraints and communication protocols require that constraints must always be met, i.e., it is not acceptable to meet the constraints "97% of the time." Our emphasis is thus on non-stochastic models, and not on queueing networks and quantitative performance evaluation (e.g., [Lazowska et al. 84]).

The expressive power of a specification language greatly affects the complexity of the formal methods for which it is the base. For example, consider a specification language for circuits that will be implemented using a microcontroller. Simple compilation techniques may be all that is needed to synthesize the design from a sequential specification. If the specification language contains constructs for specifying parallel behavior then the synthesis algorithms will need to serialize (i.e., schedule) the behavior before compilation or at run-time. The ability to express complex timing relationships will thus create complex (and possibly intractable) problems for the formal methods that will use the specification. In the first half of this dissertation, we refrain from making many restrictions to our domain of applicability because we are interested precisely in identifying the characteristics of an expressive specification language and wish to explore some of the problems that design automation tools would then have to face.

2.1.3 Designer's Needs

Some specification languages are used primarily as interchange formats or internal representations. They are often concise and structured so as to improve design tool efficiency. Other specification languages contain "syntactic sugar" to help make design specifications easier to use by human designers. Specification writers want specification languages to be natural and esthetically pleasing. Ideally, the language will be tailored to the designer's specific application area. Users may not need all of the expressive capabilities of a specification language, and they will not want to be encumbered as a result of complexities that are not relevant to their designs.

One approach to solving this problem is to have a general and expressive specification language that is augmented with application specific compilers to provide natural interfaces for designs and timing constraints that are typically of interest. These interfaces will have a more restrictive semantics, but the underlying representation can be used (by experienced designers) to express more complex functional and temporal relationships when necessary.

It is also of critical importance that a design representation support user validation. Checking that what was specified is what was desired is the first step in verifying a design and cannot be automated. A simulator provides the user with the capability to try out the circuit and make sure it behaves as expected (at least for a subset of all possible inputs). Although simulation has traditionally been used to validate design implementations, at higher levels of abstraction it is a valuable tool for prototyping and for "debugging" specifications. Most specifications are created iteratively through a process of refinement. If users can only simulate synthesized designs, they are forced to understand and work with "bugs" at low levels of abstraction even if the problems are a result of errors in the high-level specification.

Abstraction is not the only effective way to manage complexity. Specification languages need to be modular and hierarchical so that designers can decompose large designs into smaller subcomponents. Hierarchy and modularity help manage complexity and facilitate the reuse of design specifications. For example, consider a design that uses several identical subcomponents. Designers should be able to specify the subcomponent once and instantiate four instances in their design. They should not have to duplicate and rename the individual elements of their specification.

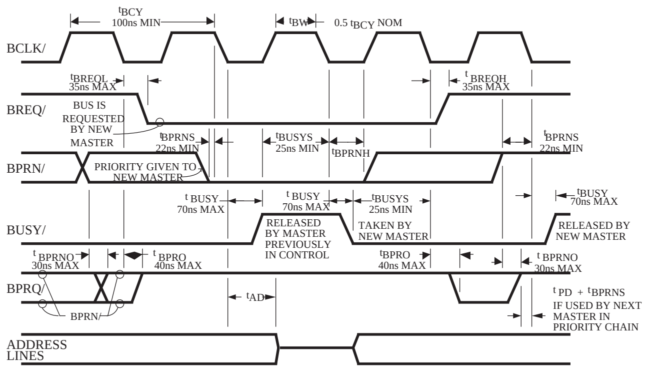
Finally, designers want to be able to use their validated formal specifications and subject them to a variety of analysis tools for synthesis and verification. Unfortunately, there exists a natural trade-off between the expressive power of specification languages, and the amounts of design automation that can be provided. Designers would like to be able to express complex systems and perform complex analysis, but often must give up one of the two.

2.2 Related Work

In this section, we present an overview of the different specification paradigms and examine some of the more well known representations. Since timing relationships usually describe separation times between circuit events, the *event-based* specification paradigm is of particular interest. The premise being that timing constraints are more easily described using an event-based representation, because the events being constrained are elements of the specification. The event-based paradigm will be discussed in further detail in the next section. There are, however, many other specification paradigms that can be used to specify timing behavior.

2.2.1 Timing Diagrams and Tables

Designers typically use timing diagrams and tables to specify timing relationships. Timing diagrams are easy to understand because they correspond to “execution snapshots” and the time axis is explicit. Transition times are not fixed, rather, timing constraints specify how "close" two events can be pushed together, or how "far" they can be stretched apart. More complex timing requirements can be specified by annotating the diagram with a textual description of the constraint. Since most timing constraints specify separation times between ordered signal transitions, tables are often used to specify the required minimum and maximum separation times. Figure 2.1 contains an example specification for the Intel Multibus, as described by the bus's documentation.



Parameter	Minimum	Maximum
XACK	0 µs	8 µs
CMD	100 ns	TOUT
ID	0 ns	100 ns
XACKA	IAD + 50ns	1500 ns
XACKB	5 ns	8 µs
AD	0 ns	
INTA	250 ns	
CSEP	100 ns	
BREQL	0 ns	35 ns
BREQH	0 ns	35 ns
BPRNH	5 ns	
BPRNS	22 ns	
BUSY	0 ns	70 ns
BUSYS	25 ns	
BPRO	0 ns	50 ns
BPRNO	0 ns	30 ns
CBRQ	0 ns	60 ns
CBRQS	35 ns	

Figure 2.1: Interface constraints for the Intel Multibus expressed using an annotated timing diagram and an accompanying table of separation times.

Specifications of this nature are not formal. They are used primarily by designers to communicate and document the temporal aspects of their designs. *Timing diagram editors* have recently been developed to help designers create timing diagrams. These editors typically are commercial products (e.g., *dV/dt* available for DOS and Macintosh [Doc 89]) or are part of in-house CAD systems (e.g., BNR's *Shadow*). These editors typically operate like spreadsheets, and often provide some form of constraint propagation and consistency checking. For example, if two signal transitions are constrained relative to a common transition, it is possible that a constraint between the two signals could result in an inconsistency. Timing diagram editors have also been developed for use as an interface to circuit testers [Arnold 85, Lai 83].

Timing diagrams constrain circuit behavior in a specific context, i.e., they represent one possible execution, assuming that a particular sequence of signal transitions occur. In real circuits, many different sequences can occur as a result of different inputs or different transition times, and thus multiple diagrams are needed in order to specify a complete system (e.g., a diagram for bus arbitration, bus read, bus write, etc.). The composition of these individual diagrams specifies all possible circuit behavior. Additional information is needed to describe when the circuit's behavior will correspond to that of a specific diagram (e.g., How are the diagrams interconnected? Which situations correspond to which parts of the diagrams?).

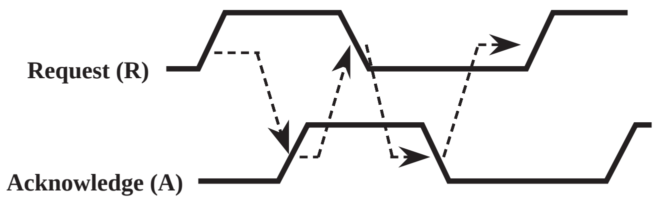
Formalized timing diagrams address these problems by providing a more complete specification language in which the interconnection between timing diagrams can be specified. Figure 2.2 contains two interconnected diagrams that together formally specify the master read operation on the Intel Multibus. These timing diagrams were produced by the editor WAVES, which is a part of an interface synthesis tool that uses formalized timing diagrams as an input specification language [Borriello 88b]. Formalized timing diagrams have also been used in timing verification to specify timing constraints which should be checked during simulation [Khordoc et al. 91].

Even though few existing tools use timing diagrams as a specification language, the paradigm has the potential to be very well accepted by designers. However, many challenging user-interface problems will need to be addressed in order to allow these languages to be expressive without sacrificing the simplicity of the basic paradigm. Designers currently specify complex timing relationships by annotating diagrams with informal text. These constraints need to be formally specified, but there are many complex timing relationships that cannot be specified using the languages of [Borriello 88b] or [Khordoc et al. 91]. These languages and their underlying internal representations lack the necessary expressive power. Furthermore, timing diagrams specify signal waveforms, not circuit functionality or structure. As such, they provide a level of abstraction that is appropriate for describing a circuit's interface (they encourage working at this higher level of abstraction) but they do not constitute a complete specification language.

Figure 2.2: Two timing diagrams from [Borriello 88b] that specify the master read operation on the Intel Multibus: *"The top diagram is the synchronous arbitration sequence while the bottom diagram is the asynchronous data transaction. The diagrams are linked at the events labeled A, B, and C. The links are used to specify common points in time across the two diagrams so that the sequences of events can be merged. Start (S) and end (E) events for the sequence are also specified. Labels on two of the waveforms (e.g., (NOT Address)) specify the flow of data values across the interfaces."*

2.2.2 Formal Logics and Related Algebras

Temporal Logics are a specification language based on first order logic with special logical operators for reasoning about time [Pneuli 77, Rescher & Urquart 71]. They have been used by many different researchers for specifying and verifying hardware (e.g., [Bennett 86], [Bochmann 82], [Browne et al. 86], [Clarke et al. 86], [Dill & Clarke 85], [Fujita et al. 83], [Fusaoka et al. 84], [Malachi & Owicki 81]). Figure 2.3 contains a four phase communication protocol specified using the linear-time propositional temporal logic of Bochmann. The first four assertions state *safety properties*, i.e., "nothing bad ever happens," because they constrain the two signals from changing in a way that would not be consistent with the protocol. The last three assertions are *livenesss properties*, which state that "something good eventually happens," namely that every request will eventually be acknowledged. Most temporal logics have a temporal operator "next" which provides a discrete unit-delay model of time. These logics are used primarily for expressing relative ordering constraints because specifying exact timing requirements is tedious and in some cases not possible.



```
 R → R while ~A
~R → ~R while A
 A → A while R
~A → ~A while ~R
 R → ∇A
 A → ∇~R
~R → ∇~A
```

Figure 2.3: A four phase communication protocol and its specification in temporal logic. "$\bigtriangledown$ A" states that A will be true at some future time.

Interval temporal logics have been developed to address the problem of specifying that an action should occur in a specific time interval [Allen 83, Moszkowski 85]. They have been used to reason about MOS VLSI circuits at the transistor level [Leeser 89], but many other logics which address the problem of modeling timing behavior have recently been developed: [Alur & Henzinger 89], [Alur & Henzinger 90], [Alur et al. 89], [Coen et al. 90], [Hansen et al. 92], [Harel et al. 90], [Koymans 89], [Koymans 90], [Lewis 90], [Narain et al. 92], [Ostroff 90]. Many of these logics use a continuous model of time and emphasize realistic modeling, others use a more restricted model to obtain feasible verification procedures. Work in this area is proceeding at a very rapid pace, and readers are referred to [Alur & Henzinger 92] for a summary of some of the more recent work.

Higher order logics (e.g., HOL [Gordon 86]) are popular languages that can be used to express complex timing relationships. Unlike the temporal logics, in which time is represented implicitly, time is usually represented by an explicit variable. For example (see [Leeser 89]), consider the definition of an inverter having a propagation delay of m

time units:

$$\mathbf{invert}(In, Out, m) \equiv \forall t \cdot Out(t+m) = \neg In(t).$$

Time is explicitly quantified in this formulation. These languages are in fact quite flexible and are used extensively for formal verification. Their complexity, however, requires that proofs be constructed by hand, with the assistance of "theorem provers," which are essentially proof managers augmented with some automated reasoning capabilities. A related formalism is *Waveform Algebra* [Augustin 89], which extends Boolean algebra to include time. Waveform Algebra is applicable to a smaller subset of behavior than that of temporal or higher-order logic (which was used to formalize the algebra). It is, however, a very elegant and simple specification language which, like timing diagrams, is more natural and appealing. Another interesting language that should be mentioned is PHRAN [Granacki 86], a formal language designed to read like English.

One characteristic of all of these specification languages is their focus on signal levels. If there are m possible levels for each signal, and there are n signals, there are m^n different assignments, each of which can be thought of as defining a particular system state. These languages often describe which states are legal or should be reachable from other system states. This focus can be a problem when different timing constraints apply during different execution contexts. For example, an acknowledge signal may need to go high within 500 ns. after a request for a low priority data transfer but may need to be acknowledged much sooner when a higher priority request is handled. With respect to timing diagrams, constraints sometimes apply to only specific transitions on signals, not every transition. Identifying which specific transitions are being constrained is difficult in these languages because levels and not events are the focus of the specification.

Many of these languages are used to specify the temporal behavior of a circuit defined using another specification language. For example, *computation tree logic* (CTL) [Dill & Clarke 85], is used to specify temporal behavior with respect to a state graph which can be derived from a circuit specification. Temporal logic has been used to specify and verify the temporal behavior of Statecharts [Harel et al. 88], and Waveform Algebra has been used to annotate VHDL [Augustin et al. 88]. There are various timing relationships that cannot be specified using the different formalisms (e.g. fairness constraints are not easily specified in CTL) and there are many well known limitations to the expressiviness of propositional temporal logic (see [Wolper 81]). Comparisons of the formal expressive power of some of these logics have been made (e.g., for a comparison of branching vs. linear-time temporal logic see [Lamport 80] or [Emerson & Halpern 86]).

2.2.3 Automata and Trace Algebras

As in the case of temporal logics, simple automata models can be used for abstract temporal modeling, in which only the sequence of events is important. The formal language accepted by these automata defines the set of legal/possible system behaviors. Because legal system behaviors are often infinite, automata that accept infinite strings (e.g., ω-automata) are often needed. We refer readers to an introductory text on automata theory (e.g., [Eilenberg 74] or [Hopcroft & Ullman 79]) for a more complete discussion of the models that support only sequencing relationships. An example of their use can be found in [Katzenelson & Kurshan 86] in which several commercial communication protocols are verified. These techniques have also been used to produce finite automata models for analog circuits [Kurshan & McMillan 91].

Timed Automata are a class of finite automata defined by [Alur & Dill 90] that can express hard real-time constraints, e.g., "the acknowledgment should occur within 5 seconds." They are based on ω-automata augmented with a finite set of timers that record the passage of time. Timers can be reset by a state transition (e.g., when a request is made) and a timer's value can be compared to a time constant and this comparison can be used to constrain state-transitions (e.g., an acknowledge is only accepted if the timer shows that five seconds or less have elapsed).

The *Input/Output Automata* of [Lynch & Tuttle 89] have been extended to support more exact timing requirements by [Bestavros 90], [Merritt et al. 91], and [Lynch & Attiya 92]. One advantage of this specification language is that complex systems can be specified by composing together simpler system components. A new model for concurrent systems, *behavior finite-state machines (BFSMs)* [Leeser et al. 91] has special language constructs for specifying communication between interacting automata. Communication allows designers to work at a higher level of abstraction and is a feature that helps automata models avoid the state explosion that would otherwise occur when multiple automata (that are essentially independent) are composed and expressed using a single automata.

Trace Theory is a general framework for specification where system behavior is described by a set of *traces*. A trace is an ordered sequence of system activities (e.g., transitions on input and output wires). For example, a legal trace of the four cycle communication protocol of Figure 2.3 would be: "$\mathbf{R}\!\uparrow$; $\mathbf{A}\!\uparrow$; $\mathbf{R}\!\downarrow$; $\mathbf{A}\!\downarrow$; $\mathbf{R}\!\uparrow$; $\mathbf{A}\!\uparrow$; $\mathbf{R}\!\downarrow$; $\mathbf{A}\!\downarrow$ ..." Trace theory has been used extensively to verify asynchronous speed-independent circuits [Dill 88], [Rem et al. 83], [van de Snepscheut 85], [Ebergen 87]. There are a variety of extensions to this body of theory that handle more detailed timing requirements. One approach introduces a *fictitious clock* and a special *tick* transition that is used to measure elapsed time. The number of tick transitions that occur between two other transitions in a trace specifies the delay between the two transitions. This model is not completely accurate. For example, it is not possible to state that two transitions are separated by exactly 2 seconds; the presence of two ticks could indicate that the transitions are separated by at least 1 but no more than 3 seconds. The granularity of a tick can be changed (e.g., one tick equals .001 seconds) to obtain a suitable degree of accuracy. A more realistic approach based on a continuous model of time uses *timed traces* in which every transition is labeled with a real number. A summary of the many different timing models (i.e., quantized vs. continuous, interleaving vs. simultaneity, etc.) can be found in [Burch 92] which presents a unifying theory for trace algebras that are used to specify temporal behavior.

There are close relationships between finite automata, trace theory (algebra), and the formal logics and algebras described in the previous subsection. For example, the *tableau* *method* is a well known decision procedure for propositional temporal logic that is based on the translation of a logic formula into a finite automaton on infinite sequences. A set of traces can be viewed as a formal language which can often be specified as a regular set or a finite automata. Algebras, logics, and automata are fundamental mathematical concepts which can be used to provide a formal semantics for a specification language. Many specification languages for timing behavior are based on direct extensions of these models.

With respect to the criteria outlined in Section 2.1, several general comments about these paradigms can be made. First, these representations are quite formal and there is often an associated body of theory that can be used to solve design automation problems (e.g., language containment for verification). The expressive power of these formalisms is in many cases well understood (e.g., trace algebra cannot be used to adequately model branching time properties [Burch 92]), but more theoretical work relating the formalisms to the specification of timing behavior needs to be done. Some of these specification languages are limited with regards to their expressiviness (e.g., the fictitious clock models, and BFSMs in which only simple linear timing constraints can be specified). In some cases, the languages are clearly not suitable for use by designers, and higher levels of abstraction are needed. Some of these formalisms in fact *require* that higher level specification languages be built on top of them. For example, how does a designer specify a set of acceptable transition sequences/traces? Almost all of these languages provide very little with respect to user validation of the specification. There are, of course, some notable exceptions (e.g., the simulator *Tempura* for Interval Temporal Logic [Moszkowski 86]), and in many cases there is no reason that a validation tool could not be built, it is just that this area has yet to reach maturity.

2.2.4 Hardware Description Languages

Our next paradigm of interest is quite dissimilar, in that the specification languages are weak with regard to formality, but strong with respect to user validation. Much like programming languages are used to describe pieces of software, *hardware description languages (HDLs)* are used to create textual "code like" specifications for hardware. Many of these languages have, in fact, been derived from existing programming languages: AHPL from APL, VHDL from ADA, HardwareC from C, etc. Hardware description languages characteristically include very little support for the specification of timing requirements. Most are based on a synchronous model that limits the circuits that can be described. HDLs are typically used to generate data-flow graphs which have formed the basis of most high-level synthesis research (see [McFarland et al. 90]).

Because HDLs have been designed to express circuit functionality, temporal constraints are often simply embedded in the design specification. In order to describe the constraints on an interface, a specification that exhibits proper behavior is specified. For example, the language SLIDE [Parker & Wallace 81] has *delay* and *delay until* statements which can be used to specify the interface from a functional perspective. This seriously restricts the types of constraints that can be specified (because constraints must be functionally specified), and embedding the constraints into the specification can create problems with respect to design synthesis. For example, if a signal transition must occur after another transition, hardware that checks that this condition is met may be synthesized because the constraint was specified functionally (e.g., using a *delay until*) and is indistinguishable from required parts of the design.

Correct behavior can also be specified by writing program code to check for incorrect behavior. For example, in VHDL, a separate process can monitor the two signal transitions and report an error (during simulation) when a timing violation occurs. This approach is, however, very ad-hoc, and a more structured and organized method for expressing constraints is clearly needed.

Behavioral Synthesis with Interfaces (BSI) [Nestor 87] is an extension to ISPS [Barbacci 81] and the Value-Trace [McFarland 78] that has a higher-level construct for specifying timing constraints. Minimum and maximum separation times between system activites are specified using labels attached to program statements. A similar approach (see Figure 2.4) is used by the more recent language HardwareC [Ku & de Micheli 90].

```
{
 tag   rd, wr, op;

 /* perform tasks */
 rd:  data = read(input_port);
 op:  result = some_function(data);
 wr:  write output_port = result;

 /* specify timing constraints */
 constraint mintime from rd to op = 3 cycles;
 constraint maxtime from op to wr = 5 cycles;
 constraint maxtime from rd to wr = 10 cycles;
}
```

Figure 2.4: Timing constraints expressed using HardwareC

Since these specifications are used for synchronous synthesis, the basic unit of measurement is a clock cycle. Scheduling algorithms are used to assign each program statement to a specific control step (cycle). The algorithm used for HardwareC is quite novel in that it can handle external synchronizations and unbounded delays between system activities [Ku 91]. Of course, in both cases, the timing constraints need to be taken into account in order to produce a valid schedule. Iterative and conditional behavior are specified using hierarchy. One serious limitation is that timing constraints are not allowed to cross the hierarchy and must be local with respect to each basic block (i.e., each acyclic data flow graph). More complex constraints can be specified in languages that use some other form of annotation. For example, *VAL* [Augustin et al. 88] is an annotation language based on Waveform Algebra that can be used to specify timing constraints for VHDL.

Hardware description languages are very popular. One of their main benefits is that they usually provide a simulator which allows designers to execute their specifications and interact with their designs. This facilitates rapid prototyping and specifications can include low-level structural components (e.g., gate-level schematics of an existing chip) as well as more abstract implementation independent descriptions (e.g., a high-level algorithmic description of matrix inversion). This popularity and concerns for simulation efficiency have led to the development of language features designed specifically to support simulation (e.g., management of the event queue).

Unfortunately, some existing hardware description languages have no formal semantics, and rely on the behavior of the simulator to define the semantics. This can lead to different semantics based on different implementations/interpretations of the language's syntax. For example if two calendar events in a discrete event simulation are scheduled to occur at the 'same time' it is possible that their execution order will impact future behavior. One implementation may choose an event at random, whereas another may always use a first-in first-out scheduling priority. Uses of the specification for purposes other than simulation (i.e., synthesis and verification) clearly indicate that the formal semantics should not be solely driven by the needs of simulation.

2.3 The Event Paradigm

The event-based specification paradigm is based on the premise that timing constraints specify relationships between circuit "events" and that these events should be elements of specification languages that are used to describe temporal behavior. The representations we have described above emphasize signal values and not signal events.

With respect to circuit structure, an event is a convenient abstraction for interesting aspects of a circuit's operation. An event can be thought of as a signal transition – a change in signal value. In this case, an event is an abstraction for a meaningful change in voltage. In some cases, events are abstractions of a temporal nature, e.g., an event corresponding to a signal's transition from "invalid" to "valid" because the signal's value is sampled and interpreted as data only during particular intervals of time. At higher levels of abstraction, events can be used to specify behavior, and represent messages or sequencing constructs, e.g., "start new mode of operation."

2.3.1 Existing Event-Based Specification Languages

Including events in a specification leads, quite naturally, to graph based representations (i.e., *event graphs*) in which nodes are used to represent events, and edges are used to represent causal or temporal relationships between events. Most of the existing event graph representations are acyclic and represent one execution of a particular system behavior. Some of the representations using this paradigm include: [Borriello 88a], [Gahlinger 90], [Hayati et al. 88], [Khordoc et al. 91], [Martello et al. 90], [Sherman 88], [Zahir & Fichtner 90]. Cyclic event graphs are far less common, and have been used primarily to represent asynchronous circuits, e.g., [Chu 87], and to analyze their performance, e.g., [Burns 91].

A closely related and well known representation is the *Petri net*, which has been used extensively to model concurrent systems (see [Murata 89] or [Brauer et al. 87] for a summary). Many different models of time have been defined for Petri nets, the most popular being stochastic delays used for probabilistic performance evaluation (see [Ajmone Marsan 89]). A number of *timed Petri net* models have been developed that incorporate non-probabilistic (e.g., fixed or bounded) delays, e.g., [Ramamoorthry & Ho 80], [Ramchandani 74], [Merlin 74], [Coolahan & Roussopoulos 85], [Cohen et al. 89], [Andre 91], [Zuberek 91].

Some specification languages are based on the event paradigm but are also closely related to the paradigms previously discussed. Timing diagrams are, in fact, essentially event-based specifications – event graphs are used to provide a formal underlying semantics (e.g., [Borriello 88b, Khordoc et al. 91]). In order to capture more complicated timing constraints (e.g., ones that appear as annotations to timing diagrams) event graphs are not sufficient, and thus more powerful specification languages are needed. An approach that uses timing diagrams and *annotated event structures* is presented in [Subramanyam 90].

Real-Time Logic [Jahanian & Mok 86] is a formal logic based on the event paradigm that is quite unlike the temporal logics previously described. The logic has been used to model real time systems and software, where an event is "a temporal marker, i.e., the occurrence of an event marks a point in time which is of significance in describing the behavior of the system." The following Real-Time Logic formula specifies a response time constraint:

$$\begin{aligned} \forall i \quad @(\mathit{start\_sampling}, i) &> @(\mathit{button\_pressed}, i) \quad \wedge \\ @(\mathit{finish\_sampling}, i) &< @(\mathit{button\_pressed}, i) + 20 \end{aligned}$$

The notation $@(e,i)$ represents the time of the i^{th} occurrence of event e. This constraint states that "sampling" should start only after a button is pressed, and should complete no later than 20 time units later. The logic supports both existential and universal quantification of integer variables that are used to identify events by their occurrence index. This indexing approach is also used in the language ATCSL [Doukas 91]. The language CPA [McFarland 90] is another formal logic based on the event paradigm. It uses a fully declarative semantics to describe a large class of system behaviors. Because the semantics is declarative, CPA provides nice mechanisms for expressing functional transformations between inputs and outputs.

2.3.2 Problems with Existing Event-Based Specification Languages

Event-based representations also present unique problems, the most serious limitation being that they are often not sufficiently expressive. Many representations are well suited to describing particular types of hardware, but there are often behaviors or timing constraints that they cannot express. This is particularly problematic since in many cases the exact nature of what is and is not expressible does not correspond well with particular classes of circuits. Many representations can express most aspects of a particular circuit's timing behavior, but find that there are some specific timing constraints or functionalities they cannot express (e.g., [Borriello 88b], [Khordoc et al. 91], [Martello et al. 90]).

Another serious problem is that structure cannot be explicitly represented in the specification. Events are "logical transitions on wires," but there are a number of problems resolving the basic issue that a wire is a continuous entity, not a collection of discrete events. Most existing representations are thus ill-suited for specifying gate-level functionality since the only allowed elements are events. Structural elements (e.g., gates) that operate on signal values instead of signal transitions are not well-integrated into these representations which are focused on very specific levels of abstraction.

Existing event-graph representations have overly simplified notions of both functionality and timing constraints. Functionality is often embedded into the semantics of an event node (e.g., an event doesn't happen until all of the incident events happen) and different types of nodes are used to express differences in functionality (e.g., two different nodes, one in which out-degree represents choice and the other in which out-degree represents parallelism). Timing constraints are often syntactically represented as edges in the graph, with an underlying restrictive semantics that does not account for the complex issues surrounding the specification of timing behavior. Of course, one advantage of these restrictions is that the timing constraints are simple and easy to understand. Timing analysis algorithms can then be borrowed from graph theory, e.g., longest paths, PERT analysis, compaction, etc. ([Borriello 88b], [Khordoc et al. 91], [Martello et al. 90]).

2.3.3 Extending the Basic Model

An expressive event-based representation must contain three important elements:

- the ability to express a rich functional semantics,
- a clean integration of structural aspects into the specification, and
- a syntax to describe timing constraints between events.

We believe that these three elements can be combined into an expressive event-based representation. However, before presenting the details of a new representation that possesses these characteristics (in the next chapter) we discuss how the need to include these elements affects some of the basic tenants of event-based representations.

An event-based representation must contain a rich notion of functionality. Functionality describes how events are interrelated; how new output events arise from input events. A circuit's functionality can be described using low levels of abstraction by providing details that spell out all internal events and their interconnection. Functionality can also be described at a high level of abstraction by specifying only the behavior of the circuit with respect to its interface. This is important if the specification method is to be scalable.

To describe structural functionality, many complex issues need to be addressed: Can input events overwhelm the circuit components by arriving faster than they can be processed? What are the characteristics of the propagation delays (e.g., inertial or transport)? Is an event observable even if it really didn't change the value of a wire (e.g., 'quiet' vs. 'stable' in VHDL)? Probably the most fundamental issue is simply how much to include: multi-valued wires, bundled wires, buses, delay models, don't-cares, etc.

Many complex issues regarding behavioral functionality also have to be addressed: Can one specify a zero-time propagation delay? If two different events need to occur before an action is taken, what happens if one of the events happens twice? Are events stored or buffered? Is there a notion of local or global state? Global state may be desirable from a user's perspective but has serious implications with respect to modularity. Clearly, events can be generated in complex ways, and thus any truly expressive representation must provide a good mechanism for specifying arbitrary functionality.

Although parts of the representation may be synthesized, the representation should not be restricted due to the need for synthesis. The specification serves to describe what is desired, or what will be observed. If the representation is being used to describe an already existent piece of hardware (for simulation), a high-level representation is necessary. The representation will need to capture the full functionality of the device even if the internal structural implementation is not known. Modularity from a behavioral as well as structural perspective is thus also quite important. Some existing representations are not expressive because their development has been too closely guided by synthesis concerns. For example, in *State Transition Graphs* (STGs) [Chu 87], there can only be one rising and falling transition on each signal (e.g., one A+ and one A− in the graph). From a high-level perspective, however, there may be two unique independent contexts in which a signal should be raised, and thus two distinct events are needed. Synthesis algorithms have been the focus for the development of STGs and therefore STGs have not evolved into a general high-level representation.

Although simple timing constraints often have an obvious representation (e.g., a labeled directed edge representing the difference in time between two events) the representation of constraints between events nested within loops, forks, conditional branches, and concurrent structures requires a more comprehensive mechanism. The problem is that constraints are relationships between discrete events—not event nodes (see Section 3.2 for a clarifying example).

In order to describe timing constraints, a means of identifying the discrete events being constrained is needed. Chronological relationships can often be used to identify the discrete events involved. However, constraints may be relative to a particular execution path in a complex graph, and chronological relationships alone are not sufficient. Constraint specification must include a way of getting at this history so as to describe the context (execution path) in which the constraint applies. For example, timing diagrams often specify constraints that apply only during a particular mode of operation (in Figure 2.1, some of the constraints apply only if a particular arbitration mechanism is used).

Of course, one solution to the problem of identifying the discrete events being constrained is to have the events in the specification represent individual discrete instances of signal transitions. This approach is of limited use in a specification language since it assumes that the circuit's behavior can be statically described. This approach may have some use in validating simulations with known expected behavior [Khordoc et al. 91]. However, if circuit behavior is repetitive then the representation will be cyclic. Thus, event nodes inevitably represent more than one signal transition and the problems described above must be addressed.

Chapter 3

A New Model

We have developed a new specification language for timing behavior that meets most of the criteria and goals outlined in the previous chapter. The representation is based on the event graph paradigm with an operational semantics for specifying design functionality. Timing constraints are expressed using a formal logic that is also event-based and that enables the expression of complex timing behavior.

3.1 The Operation-Event Graph

An *operation-event graph* is a bipartite[1] directed graph $G(O, E, A)$ with dependency arcs A and nodes $O \cup E$ where O is a set of *operations* and E is a set of *events*. We refer to the components of the graph as arcs and nodes instead of edges and vertices in order to avoid the potential confusion that would otherwise arise given our choice of notation (E denotes a set of vertices, not edges). The topology of the graph is restricted such that each event has an in-degree of either 0 (the event is an external input), or an in-degree of 1 (the event is an internal output of a single operation). Events may have arbitrary out-degree.

For each event $e \in E$ there is an associated pair $\langle s, v \rangle$ where:

[1]In a bipartite graph, the nodes are partitioned into two disjoint sets (O and E in this case) and the arcs connect nodes in one set to nodes in the other set.

$s = signal(e)$ is a *signal*, and

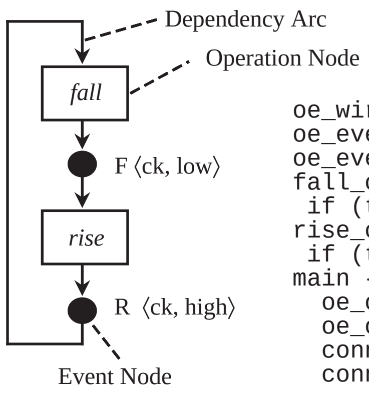
$v = value(e)$ is a *signal value*, and we restrict v to be an element of V, where

$$V = \{\textbf{high}, \textbf{low}, \textbf{dc}, \textbf{valid}, \textbf{tri}, \textbf{undefined}\}.$$

The association of a signal and a value to each event in E is technically a one-way mapping, because we allow different events to have the same $\langle s, v\rangle$ pair. This is necessary because there may be two or more unique contexts in which a signal should be raised. For example, a specification could contain two different operations: one that handles high priority requests, the other low priority requests, and both operations might need to raise an acknowledgment signal. An event may be associated with a *fictitious* signal and **undefined** signal value. This is necesary because events can represent behavioral abstractions which do not correspond to signal transitions (e.g., control events, such as an event indicating the completion of an activity, or a reset message).

Operation-event graphs are a generalization of the event-graph model described in the previous chapter. This generality is a result of the introduction of operations to specify how events are causally related to one another. In other models, this information is typically embedded into the semantics of an event node, and is not explictly defined. Instead, we associate with each operation $o \in O$ a piece of *program code*, which describes the functionality of an operation. The program code is evaluated whenever an input event occurs. The evaluation may conditionally generate output events (which usually incur some delay) and/or change internal state. An example of an operation node is a logic gate that generates an output event whenever an input event occurs. A more abstract example of an operation is one that arbitrates between two processes: the operation decides which of two output events to generate, thereby permitting one of the processes to proceed. The two events, in this case, are not logic transitions and instead represent control flow.

Our model is quite general because operations permit abstract specification of input/output behavior. For example, an operation could count the number of input events that it has seen, and generate an output event only when the number of observed events is prime. The program code, in this case, would contain an algorithmic description of how to determine whether or not a number is prime. By restricting the semantics of an operation-event graph, a model equivalent to that of a more restrictive event-graph model can be created. For example, operation program code could be restricted such that only the semantics of STGs (i.e., an output event may occur only when all inputs events have occurred) is allowed. Figure 3.1 contains a sample operation-event graph represented both graphically and textually.



```
oe_wire ck("ck");
oe_event F("F",ck,LOW);
oe_event R("R",ck,HIGH);
fall_code(oe_trigger trigger) {
 if (trigger==R) cause(F,25);}
rise_code(oe_trigger trigger) {
 if (trigger==F) cause(R,25);}
main {
  oe_operation fall("fall", fall_code);
  oe_operation rise("rise", rise_code);
  connect(fall,F);connect(R,fall);
  connect(F,rise);connect(rise,R);}
```

Figure 3.1: Graphical and textual versions of a simple single phase clock in the OEgraph representation (from a behavioral perspective). The clock has a cycle time of 50 time units and a 50% duty cycle.

3.1.1 Formalizing the Semantics

An operation-event graph is a framework which can be used to construct a specification language given a particular formalization of the syntax and semantics of the program code. The semantics should, however, be operational, in that the code essentially describes *what* an operation does with its inputs. An operation-event graph is used to provide a global view of system functionality, each operation conceptually being a small independent piece of functionality. Events interconnect operations, carry data, and define control flow. Operations thus may not have access to any global information. The program code can not contain shared global variables and operations should not have access to information such as the time of a discrete event's occurrence, the value of a signal which is not incident to the operation, or knowledge about other facets of the operationalized semantics (e.g., knowledge about simulation event queues, etc.). This is not to say that an operation cannot measure the "elapsed time" between input events or use the "value of a signal" to perform a computation. Rather, such actions are not accomplished directly by an operation's program code. They can be explicitly defined using events, i.e., an operation can implement a timer with a tick event feedback and thus keep track of the number of ticks it has seen, or an operation can know the value of a signal by having every event associated with a particular signal (wire) as an input.

The operation program code semantics is best defined by relating the syntactic elements of the code to the set of *executions* that describe all of the possible systems behaviors that could be exhibited by the operation-event graph. An execution is a potentially infinite set of *discrete events*.

One of the major points of emphasis in this dissertation is that there is a difference between a discrete event (a member of an execution) and an event node (a member of E). This difference is of fundamental importance with respect to the event paradigm and event-based specification languages for timing behavior. An event node represents an event that could occur many times during system execution, e.g., an event on a cycle in an operation-event graph. A discrete event is an actual *occurrence* of an event node, at a specific instant of time.

Formally, we associate with each discrete event d a quadruple:

$$\langle event(d), value(d), \tau(d), ancestry(d) \rangle$$

where:

$e = event(d)$ is an event, i.e., d is a discrete occurrence (an instance) of event e,

$value(d)$ is a signal value,

$\tau(d)$, a non-negative integer, is a *time-stamp* that denotes the time of d's occurrence, *ancestry*(d), a set of discrete events, is an *ancestry-stamp* used to capture information about the causality of d.

For an operation-event graph $G(O, E, A)$ with operations O, events E, and dependency arcs A, let $\mathcal{X}$ denote an arbitrary execution of G. Although we will not provide a complete and formal definition of our semantics (because we have not defined the syntax of the operation program code), we can futher clarify our semantics by describing some constraints on $\mathcal{X}$:

$d \in \mathcal{X} \Rightarrow \mathit{event}(d) \in E$, i.e., only instances of events in E may occur,

$d \in \mathcal{X} \Rightarrow$

$$\begin{aligned} (\mathit{value}(\mathit{event}(d)) \neq \textbf{valid} \quad &\Rightarrow \quad \mathit{value}(d) = \mathit{value}(\mathit{event}(d))\ ) \ \wedge \\ (\mathit{value}(\mathit{event}(d)) = \textbf{valid} \quad &\Rightarrow \quad \mathit{value}(d) \in \{\textbf{high}, \textbf{low}, \textbf{valid}\}) \end{aligned}$$

I.e., event nodes and their discrete occurrences must both be associated with the same signal value except that discrete occurrences of **valid** events can be associated with **high** or **low** transitions—see Section 3.1.3 for more information about the significance of **valid**,

$d \in \mathcal{X} \Rightarrow \forall d' \in \mathit{ancestry}(d),\ \tau(d) \geq \tau(d')$, i.e., d cannot occur before any of its ancestors occurred.

$d \in \mathcal{X} \Rightarrow \forall d' \in \mathit{ancestry}(d) \quad \langle o, \mathit{event}(d)\rangle \in A \Rightarrow \langle \mathit{event}(d'), o\rangle \in A$ i.e., the ancestors of d are occurrences of events that are incident to the operation that generated d.

Our model of time is now apparent from our definition of $\tau(d)$. The time at which a discrete event occurs is not an infinite precision real, but instead is an integer. Furthermore, multiple events may occur at the same time, but because an execution is an unordered set of discrete events, it is not known which occurred first. We believe that this model of time is appropriate because a granularity problem also exists in the physical world which we are modeling, i.e., at some level of detail it is not possible to determine which of two discrete events actually occurred first. We believe that a total order with respect to chronology is thus not realistic, nor do we wish to deal with true simultaneity. Using integers (as opposed to a dense set) is efficient and helps to minimize the amount of space used to store a discrete event. Operations execute atomically and instantaneously due to the occurrence of a single input event. If two input events occur at the same time, two evaluations of the operation will occur, and each evaluation will have a different *triggering event* (which of the two discrete events *triggers* the first evaluation is deliberately not defined).

We have developed the concept of *event ancestry* to address the issue of specifying timing constraints that are dependent on an execution context. This is the motivation for associating an ancestry-stamp with each discrete event. Whenever an operation decides to generate a discrete event, the new event has as *immediate ancestors* the most recent discrete occurrences of the input events named as ancestors. Intuitively, the immediate ancestors of a discrete output event are the discrete input events that were used to determine that the output event should be generated. For example, in Figure 3.1, discrete occurrences of F have discrete occurrences of R as ancestors, and vice-versa.

In order to be able to decompose operations into more primitive elements, operations need to be able to capture ancestry information internally. Consider an operation that stores the signal value of an input discrete event on a queue (of state variables). When the value is removed from the queue and "attached" to an output discrete event (see the discussion of **valid** in the next section) the output event's immediate ancestor should be the input discrete event that originally caused the value to be placed on the queue. The event that caused its removal from the queue should, of course, also be an ancestor, as it too was responsible for generating the output event.

Note that both the time-stamp and the ancestry-stamp are used to *define* the semantics of an operation, they are not used to specify functionality. An operation cannot decide what to do based on the time-stamp or ancestry-stamp of one of its inputs.

3.1.2 OEgraphs: A Textual Specification Language

We have developed a textual specification language, *OEgraphs*, that is an instance of an operation-event graph. The single phase clock in Figure 3.1 was textually specified using OEgraphs. We make a distinction between our specification framework, the operation-event graph, and our textual specification language, OEgraphs. Both the language and the framework were developed in parallel, and in some sense the framework simply isolates and identifies the unique and important elements of our specification philosophy.

OEgraphs are based on the programming language C++ augmented with data structures and other language constructs that support the operation-event graph specification paradigm. We do not intend to present a complete syntax and semantics for OEgraphs; our examples should, however, be quite readable. The language contains declarative constructs used to specify the operation-event graph's components (e.g., types `oe_event` and `oe_operation` for events and operations). These declarations have arguments which specify the associated values, e.g., an event's signal value, or an operation's program code.

The program code for each operation consists of a procedure that is invoked each time an incident event occurs. The *cause* statement is used to generate events (i.e., see Figure 3.1). Ancestry information can optionally be specified for each cause. By default, i.e., if ancestry information is not specified (as in Figure 3.1), the most recent occurrences of all input events are immediate ancestors. Ancestry information can also be captured via the use of special state variables which have ancestry stamps.

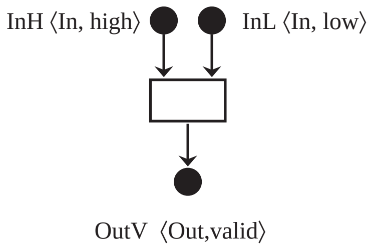
The delay information for each discrete event that will be generated by an operation is specified either by a fixed non-negative integer or an *unspecified* delay value. The delay is relative to the time of the triggering event (the discrete event that caused the operation to be evaluated). Unspecified delay values may be bounded, and in this case, a distribution function can also be specified (e.g., the delay is uniformly distributed between 10 and 20 nanoseconds). Timing constraints can be used to specify proper temporal behavior of circuits containing operations with unbounded unspecified delays. Such specifications are not particularly well suited to simulation, e.g., the user must provide the delay value every time one is needed, but these specifications are useful for specifying functional behavior which will be temporally constrained using a different formalism, namely the constraint logic of Section 3.2.1.

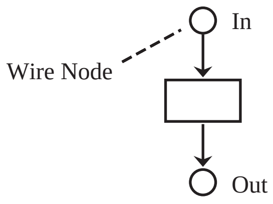
3.1.3 Incorporating Structure

Designers need to be able to specify the components of a design using different levels of abstraction. If a single specification language can capture a design at all of these different levels of abstraction, many benefits, both tangible and intangible, accrue. One significant benefit is that synthesis can be viewed as a transformational process; behavioral specifications are transformed, through a series of steps, into structural specifications. In Section 2.3, we argued that a clean integration of structure into event-based specification languages was thus needed, and that representations should allow structural components to be specification elements.

We believe that operation-event graphs provide a good formal basis for expanding the expressiveness of event-based specification languages. We have included and integrated structure into the OEgraph specification language. Operations are allowed to operate on signals, e.g., to have signals as inputs and outputs. We have defined a new node type, a *wire* which is really a convenient syntactic extension. Operation-event graphs are quite capable of *modeling* structure. In OEgraphs, we simply extend the syntax of an operation's program code to operate on signal values. This is, in some sense, merely a *perspective*, a way of encapsulating and viewing the structure. Figure 3.2 shows behavioral and a structural perspectives of an inverter.

Both specifications make use of the special properties of the signal value **valid** (although this is explicit only in the behavioral version). We allow event nodes to be associated with a transition to **valid** and their discrete events to be associated with transitions to either **high** or **low** depending upon the dynamic behavior of the operations generating the events (i.e., statically **valid**, at run-time either **high** or **low**). This facilitates the notion of *data*, in that operations may obtain the signal value for a dis-





```
oe_wire In("In");
oe_wire Out("Out");
oe_event InH("InH",In,HIGH);
oe_event InL("InL",In,LOW);
oe_event OutV("OutV",Out,VALID);
inv_code(oe_trigger trigger) {
  if (trigger==InH) cause(OutV,10,low);
  if (trigger==InL) cause(OutV,10,high);}
// or, cause(outV,10,not(valueOf(trigger));
main {
  oe_operation inv("inv", inv_code);
  connect(InH,inv);
  connect(InL,inv);
  connect(inv,OutV);}
```

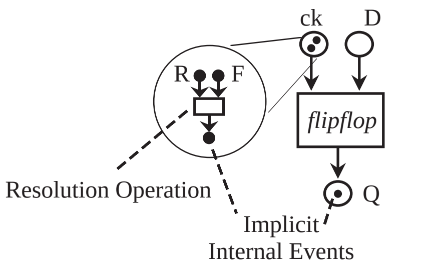
```
oe_wire In("In");
oe_wire Out("Out");
inv_code(oe_trigger trigger) {
  cause(Out,10,not(In);}
main {
  oe_operation
    inv("inv", inv_code);
  connect(In,inv);
  connect(inv,Out);}
```

Figure 3.2: Two different specifications for an inverter, the left specification is behavioral and operates on events, the right specification is structural and operates on wires.

crete input event and decide what value a discrete output event should then assume. For example, the program code for the inverter could also be specified as noted by the textual comment in the left specification of Figure 3.2. Note that we purposely have made no distinction between data and control, i.e., events can represent and carry both data and control information. This will allow optimization and synthesis algorithms greater flexibility.

Conceptually, an operation with a structural input signal s should have all of the events that occur on s (i.e., $\{e \mid signal(e) = s\}$) as inputs. The operation should not distinguish between the different input events; rather, all of the events should be treated the same way, because it is the value of the signal that is important, not the occurrence of a specific named event on the signal. Likewise, an operation with a structural output should be the only operation generating events for a particular signal.

Thus, although operation-event graphs do not contain structure (signals are not nodes in the graph), our specification language OEgraphs allows specifications to "look like structure." In essence, the presence of a wire merely indicates the presence of an implicit internal event—either an input or an output, e.g., see Figure 3.3.



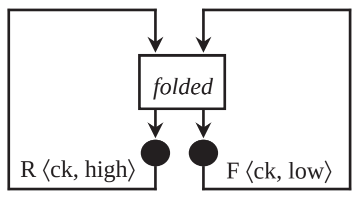
```
oe_wire ck("ck");
oe_wire D("D");
oe_wire Q("Q");
ff_code(oe_trigger trigger) {
 if (trigger==ck && ck==HIGH)
   cause(Q, uniform_delay(5,10),valueOf(D);}
main {
  oe_operation flipflop("flipflop", ff_code);
  connect(ck,flipflop);
  connect(D,flipflop);
  connect(flipflop,Q);}
```

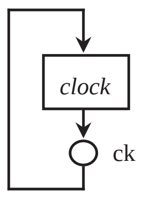
Figure 3.3: A clocked edge-triggered D flipflop represented using OEgraphs. Note that the propagation delay is uniformly distributed between 5 and 10 time units. The flipflop operation does not hold state; Q is set to the value of D on every rising clock edge. The transitions on the clock (wire ck) are event nodes (e.g., F and R from Figure 3.1) but the flipflop has ck (and D) as an input and doesn't care about the specific event nodes that "make up" the wire.

Every wire has a *physical model* which is used to determine how events affect wires (e.g., a resolution function in VHDL). Structural details are essentially handled by changing the structure of the *resolution operation* that collects all of the events for a particular wire, e.g., the operation in Figure 3.3 that collects the events R and F. We define $\mathit{resolve}(s)$ to be the name of the internal event that collects all of the events occurring on s (for when s is an input wire to an operation), and $\mathit{output}(s, o)$ to be the name of the internal event that is used when operation o has the wire s as an output.

Operations having wire inputs and outputs represent functionality from a structural perspective. Operations having event inputs and outputs represent functionality from a behavioral perspective. Mixed perspectives are also possible, e.g., an operation having wire inputs carrying data and event inputs signalling flow of control. Figure 3.4 contains two representations of our simple single phase clock (Figure 3.1). We first "fold" our clock into a single operation, and then since all of the events on the clock are inputs and outputs of this single operation, we can transform the operation into a purely structural specification using only wires.

Of course a good specification language for structure will need to include support for





```
oe_wire ck("ck");
oe_event F("F",ck,LOW);
oe_event R("R",ck,HIGH);
f_code(oe_trigger trigger) {
 if (trigger==R) cause(F,25);
 if (trigger==F) cause(R,25);}
main {
  oe_operation folded("folded", f_code);
  connect(folded,F);connect(folded,R);
  connect(F,folded);connect(R,folded);}
```

```
oe_wire ck("ck");
ck_code(oe_trigger trigger) {
 cause(ck,25, not(ck));}
main {
  oe_operation clock("clock", ck_code);
  connect(clock,ck);
  connect(ck,clock);}
```

Figure 3.4: Structural transformations of the single phase clock in Figure 3.1

a variety of structural properties: bundled wires (busses), transport and inertial delay models, etc. Some of these issues have been addressed by our specification language. However, our emphasis has been on expressing temporal and not structural properties. We do believe that operation-event graphs are a framework within which these issues can be suitably addressed. For example, in most cases, the effect of an event on a signal corresponds exactly to the transition that the event represents. For a bus, however, this is not necessarily the case. With two or more operations driving a bus, an event asserting a tri-state value may leave the bus connected to ground (and not disconnected) because another operation connected to the bus keeps it low. Normally, any wire that has an indegree of more than one is a bus (i.e., two or more operations are driving the wire via multiple internal events). Two or more events on the same wire may or may not represent a bus, depending upon whether or not synthesis transforms the events into wire outputs of one or several physical components (represented as operations). In OEgraphs we keep track of this information using partition labels which can be attached to events. Two events (on the same wire) with different partitions represent events interacting on a bus.

In summary, OEgraphs support structure by allowing users the freedom to write structural specifications which are then appropriately interpreted within the context of the operation-event graph model. Users are able to work at their desired level of abstraction. Events can be generated on wires without the need to name them explicitly.

3.2 Timing Constraints

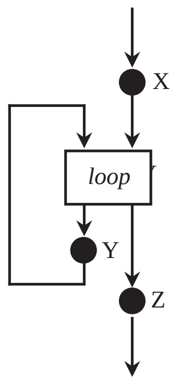
The semantics of operation-event graphs is operational. Operations describe outputs by explicitly stating how the inputs are transformed. The relationships between the various parts of the specification are explicitly represented using events and dependency arcs. If the specification consisted merely of a list of requirements it would be more difficult to view the design as a collective entity. The operational semantics by its very nature facilitates design validation. Specifications can be executed and the response to a given set of stimulus can be easily computed because the system's functionality is operationally specified.

However, timing constraints by their very nature are denotational[2]. Constraints are annotations which further constrain the temporal behavior of the system, i.e., they specify acceptable input/output behavior without describing how the inputs are transformed into the outputs. Thus, constraints are typically expressed using formalized timing diagrams, tables of separation times, or other annotations (e.g., labeled edges between event nodes in event graphs). Complex timing relationships can be difficult to express due to the presence of complex control structures, i.e., loops and conditional branches, coupled with the repetitive nature of these systems (because event nodes represent events that may occur multiple times). The fundamental issue that must be addressed in order to specify a timing constraint is that the discrete events being constrained must be identified (timing constraints are relationships between discrete events—not event nodes).

One approach to this problem is to restrict the semantics of constraint specification so that specifying event nodes implicitly identifies the discrete events being constrained. For example, consider the operation-event graph fragment shown in Figure 3.5. Three

[2]Temporal behavior can, of course, be specified by describing the set of all possible system behaviors using an operational semantics. This is often cumbersome and not nearly as natural as providing a description of the required behavior.

of the event nodes (X, Y, and Z) have been identified by name. The event Y is on a loop, and thus there will potentially be many more occurrences of Y than of either X or Z during any execution of the system. The restricted semantics might assume (e.g., as in [Borriello 88b]), that a constraint "from X to Y" relates X to the first Y occurring in the loop, and that a constraint "from Y to Z" relates the last Y occurring in the loop to Z.



```
oe_event X("X");
oe_event Y("Y");
oe_event Z("Z");
loop_code(oe_trigger trigger) {
 int count;
 if (trigger==X) count=0;
 if (trigger==Y) count++;
 if (count < n) cause(Y) else cause(Z);}
main {
  oe_operation loop("loop", loop_code);
  connect(loop,Y);connect(loop,Z);
  connect(X,loop);}
```

Figure 3.5: An operation-event graph fragment, the event Y is on a loop.

An alternative approach relies on indexing. Each occurrence of an event has an index, and this index can be used to specify which discrete events are being constrained. In this case, (e.g., Real-Time Logic [Jahanian & Mok 86], or ATCSL [Doukas 91]) one could specify a constraint "from X to the 2^{nd} occurrence of Y in the loop" by constraining the i^{th} occurrence of X with the $((i-1)n+2)^{\text{th}}$ occurrence of Y where n is the fixed iteration count for the loop ($n \geq 2$). However, if the number of Y events that can occur during every execution of the loop is not fixed (e.g., a while loop) then indexing is of limited value, because the indices themselves do not adequately serve to identify which discrete events should be constrained. Real-Time Logic and other languages based on indexing can sometimes express constraints such as this but the constraints are difficult to construct because only indices can be quantified, and the constraints are thus very complicated.

Both methods seriously restrict the types of timing relationships that can be specified because neither provides an expressive mechanism for identifying which discrete events are being constrained. For example, existing specification methods often introduce a special semantics to handle timing constraints for synchronous systems, e.g., a special unit of delay, the "cycle," is needed because the basic specification mechanisms are not powerful enough to specify synchronous constraints.

3.2.1 Event Logic

Our solution to the problem of expressing more complicated timing relationships is to use *event-logic*, a first order logic which relies on the quantification of discrete events, and not the quantification of time variables (as in HOL), or event indices (as in Real-Time Logic). The essential elements of the logic can best be illustrated with a simple example:

$$\forall d_1\, \forall d_2 \quad \tau(d_1) > \tau(d_2) \quad \Rightarrow \quad \tau(d_1) - \tau(d_2) > 10$$

This timing constraint states that no two discrete events occur within 10 time units of one another, unless they occur at the same time. Note that $\tau(d)$ is a function that represents the time of occurrence of the discrete event d. Similar functions exist for $event(d)$, $value(d)$, and $ancestry(d)$. Thus, we could write a constraint stating that the rise to fall delay for a behaviorally specified clock (with events R and F denoting rising and falling edges, as in Figure 3.1) should be exactly 25:

$$\begin{aligned}&\forall r\, \forall f \;\; event(r) = R \;\wedge\; event(f) = F \;\Rightarrow \\ &\quad (\, (\, \tau(f) > \tau(r) \;\wedge\; \neg\exists f_2\, (event(f_2) = F \;\wedge\; \tau(f_2) > \tau(r) \;\wedge\; \tau(f_2) < \tau(f)) \;\,) \Rightarrow \\ &\qquad \tau(f) - t(r) = 25)\end{aligned}$$

The constraint is complicated because it specifies a separation time between a rising edge r and the very next falling edge f (i.e., the second line of the constraint ensures that f occurs after r, and that f is the very next falling edge—the constraint does not apply if there exists another occurrence of F in between r and f).

Discrete events are identified by relating the events either chronologically or causally with one another, so as to specify a context (e.g., f is the next falling edge after r) within which the constraint applies. Constraints usually require a separation time between two of the quantified events. Causality is an informal relationship between a discrete event x and another discrete event y that occurred because of x, i.e., x directly or indirectly *caused* y. Formally, this relationship is specified using a new predicate, *ancestor*, the transitive closure of the discrete event's immediate ancestors (recall that the immediate ancestors of a discrete event d are members of the set $\mathit{ancestry}(d)$). An ancestor of a discrete event is any previously occurring discrete event that led to the generation of its descendant through its effect on a series of operations. We use the binary relation $x \preceq y$ to indicate that x is an ancestor of y:

$$x \preceq y \quad \Leftrightarrow \quad x \in \mathit{ancestry}(y) \ \vee \ (\exists z \ z \in \mathit{ancestry}(y) \ \wedge \ x \preceq z)$$

Both event-logic and operation-event graphs can be used to describe system behavior, i.e., to specify the possible set of executions and the discrete events they contain. For example, the statement:

$$\forall x \ \ \mathit{signal}(x) = \mathit{In} \ \Rightarrow \ (\ \exists y \ (\mathit{signal}(y) = \mathit{Out} \ \wedge \ \tau(y) = \tau(x) + \mathit{10} \ \wedge$$

$$(\mathit{value}(y) = \textbf{high} \ \wedge \ \mathit{value}(x) = \textbf{low}) \ \vee \ (\mathit{value}(y) = \textbf{low} \ \wedge \ \mathit{value}(x) = \textbf{high}) \ ) \ )$$

is a structural specification for the inverter of Figure 3.2. The behavioral specification would not refer to the signals, and would rely only on the events, i.e.,

$$\forall x \ \ (\mathit{event}(x) = InH \ \vee \ \mathit{event}(x) = InL) \ \Rightarrow \ (\ \exists y \ (\mathit{event}(y) = OutV \ \wedge \ \ldots$$

One could argue that both specifications are not yet complete; they specify the correct output response, but do not disallow incorrect behavior (e.g., other transitions on the signals). Our point, however, is that event-logic could perhaps be used by itself as a specification language, replacing the need for operations.

We have not formally examined the expressive power of event-logic, nor have we formally explored its relationship to other logics. However, it appears to be quite expressive, and is capable of handling a wide variety of temporal relationships. We have found that most complex databook annotations can be specified. One limitation is discussed in Section 3.3.

The logic is quite elegant, but quantifiers can appear anywhere, and complex constraints are unfortunately difficult to read. This is a general characteristic of denotational specification languages, in which specifications inevitably consist of a large number of statements that relate the outputs to the inputs. Unfortunately, the interrelationships between the statements are often difficult to grasp, and the overall required functionality is hidden by the specification. We believe that supporting both an operational semantics for functionality and a denotational semantics for constraining temporal behavior allows users the flexibility to work with both methods of specification. Thus, in many ways, the event logic is too expressive with respect to our needs. Furthermore, the logic presents a number of difficulties with respect to user validation (these will be discussed in Chapter 4).

3.2.2 Restricted event-logic for OEgraphs

Our textual specification language OEgraphs uses a timing constraint syntax that is a highly restricted form of event-logic. Constraints are specified by:

1. *Naming* the discrete events involved in the constraint, then
2. Specifying the *Context* (identifying the particular events in a particular relationship to one another), and finally,
3. Specifying the *Requirement*—the timing relationship that must hold.

The restricted event-logic formula has the following form:

$$\underbrace{\forall x_0\, \forall x_1\, \ldots\, \forall x_{n-1} \left(\bigwedge_{i=0\ldots n-1} id(x_i, r_i) \right)}_{Naming} \Rightarrow (Context \Rightarrow Requirement)$$

Where x_i denotes a quantified discrete event, and r_i denotes a *restriction* (defined below, as is the function id).

The naming of events corresponds to a series of universal quantifications of discrete events and a conjuncted series of quantifier-free formulas which further constrain the identity of each quantified event. Universal quantification does not in any way identify the events being constrained. *Naming* the events can be thought of as a process in which the identities of the events are established by static properties, e.g., specifying that the quantified event x_i represents an occurrence of the event R (e.g, $event(x_i) = R$), or represents a high transition (e.g., $value(x_i) = \textbf{high}$). Formally, we associate with each quantified discrete event a *restriction* which can be an event, a signal, a signal value, or a conjunction or disjunction of restrictions. Each restriction is used to further constrain the identity of each quantified event. The function *id* maps a quantified discrete event (x_i) and its associated restriction (r_i) to an event-logic formula:

$$id(x_i, r_i) \equiv \begin{cases} event(x_i) = r_i & \text{if } r_i \in E \\ event(x_i) = resolve(r_i) & \text{if } r_i \in \{signal(e) \mid e \in E\} \\ value(x_i) = r_i & \text{if } r_i \in V \\ id(x_i, r_1) \wedge id(x_i, r_2) & \text{if } r_i = r_1 \wedge r_2 \\ id(x_i, r_1) \vee id(x_i, r_2) & \text{if } r_i = r_1 \vee r_2 \end{cases}$$

After quantifying the events, the correct temporal behavior is specified via a formula which has been divided into two parts (the *Context* and the *Requirement*) so that constraints are more easily written and understood. If the *Context* is satisfied for a particular instantiation of discrete events, the *Requirement* must also be satisfied. (e.g., *Context* $\Rightarrow$ *Requirement*). Both expressions are quantifier free, but in order to capture much of the expressibility of event-logic we define the following three relations (which implicitly quantify events) which can appear in either formula. Let x and y be discrete events and let r be a restriction associated with x. The formal semantics of each relation is described via event-logic:

$mra(y, r, x)$ to test whether y's most recent r ancestor is x:

$$x \preceq y \;\wedge\; \neg\exists z\, (id(z, r) \;\wedge\; z \preceq y \;\wedge\; \tau(z) > \tau(x))$$

i.e., x is an ancestor of y and no other r ancestors of y occur after x.

$pco(y, r, x)$ to test whether the previous chronological occurrence of an r with respect to y is x:

$$x \neq y \ \wedge \ \tau(x) \leq \tau(y) \ \wedge \ \neg\exists z \, (id(z, r) \ \wedge \ \tau(y) > \tau(z) \ \wedge \ \tau(z) > \tau(x))$$

i.e., x occurs before y and no other r events occur in between x and y [3].

$nco(y, r, x)$ to test whether the next chronological occurrence of an r with respect to y is x:

$$x \neq y \ \wedge \ \tau(x) \geq \tau(y) \ \wedge \ \neg\exists z \, (id(z, r) \ \wedge \ \tau(x) > \tau(z) \ \wedge \ \tau(z) > \tau(y))$$

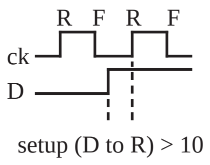
i.e., x occurs after y and no other r events occur in between x and y.

These three relations encapsulate, in a more efficient and compact form, concepts which are essential for timing constraint expression. These relations can appear anywhere in the context or requirement and may also be negated. This syntax is more closely aligned with our original goal of providing a logic for the formal description of timing (and not functional) behavior. Discrete events are identified by their properties (e.g., $event(f) = F$) and by contextual relationships of a chronological or causal nature (using *nco*, *pco*, or *mra*). Once the events have been suitably identified, the timing constraint is then stated.

3.2.3 Examples

Figure 3.6 contains the specification of the simple setup constraint for the D input to the flipflop of Figure 3.3. Most constraints, like this one, have a simple semantics (e.g., setup and hold) and parameterized subroutines for constraint specification can be defined using OEgraphs. Other higher-level specifications (e.g., timing diagram editors [Borriello 88b])

[3] We also allow x to occur at the same time as y. Note that if multiple events occur at the same time it is possible that multiple x events will satisfy the relation, i.e., if more than one event is tied for the role of previous occurrence (or most-recent or next occurrence) then all of the events satisfy the relation.



... inserted into main{} of D flipflop

```
oe_constraint setup("setup");
discrete_name D0("D0",D);
discrete_name R0("R0",R);
setup.quantify(D0,R0);
setup.context(nco(D0,R,R0));
setup.require(timeOf(R0)-timeOf(D0) >10);
```

Figure 3.6: A setup constraint for the D input to the flipflop in Figure 3.3. The textual representation is based on the restricted event-logic discussed in Section 3.2.2. In the restricted logic, this constraint would be expressed as:

$$\forall d_0 \, \forall r_0 \, (event(d_0) = D \wedge event(r_0) = R) \Rightarrow (nco(d_0, R, r_0) \Rightarrow \tau(r_0) - \tau(d_0) > 10)$$

and in the unrestricted logic the *nco* would be expanded:

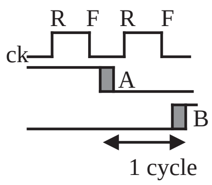
$$\forall d_0 \;\; \forall r_0 \;\; (event(d_0) \; = \; D \wedge event(r_0) \; = \; R) \; \Rightarrow \; ((r_0 \neq d_0 \;\; \wedge \;\; \tau(r_0) \geq \tau(d_0) \;\; \wedge \; \neg\exists z \; (event(z) = R \; \wedge \; \tau(r_0) > \tau(z) \; \wedge \; \tau(z) > \tau(d_0))) \Rightarrow \tau(r_0) - \tau(d_0) > 10)$$

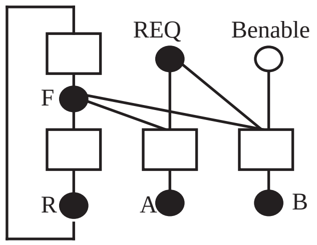
could also be integrated with this representation providing a more user-friendly syntax for (less complicated) constraint specifications.

Of course, some constraints which appear to be simple in nature actually have a complicated semantic meaning. For example a constraint stating that "two events occur one cycle apart" is subject to many interpretations. Timing diagrams and tables attempt to convey the semantics of such constraints but are informal and often ambiguous. Our representation allows such constraints to be formally specified.

Figure 3.7 contains a constraint that states that events A and B are required to occur exactly one cycle apart whenever A and B occur from the same request event (REQ). For some request events the B event is not generated and the A event should not be constrained. The constraint is also complicated by the fact that A and B occur synchronously to the falling edge of the clock (F) and may occur anywhere within the shaded region shown in Figure 3.7. Requiring an exact separation time between A and B would thus be incorrect. If this constraint in a different representation consisted of an edge from A to B labeled "one cycle," then the formal semantics would be hidden because what should happen when there are no B events would be very unclear.

Our next example provides two different formal semantics for the informal statement "This circuit can be clocked at 10 to 20 Mhz." Figure 3.8 contains three possible clock waveforms, and two possible constraints. The first version does not require that the





```
...
oe_constraint cycle_apart("cycle_apart");
discrete_name A0("A0",A);
discrete_name B0("B0",B);
discrete_name F0("F0",F);
discrete_name F1("F1",F);
discrete_name REQ0("REQ0",REQ);

cycle_apart.quantify(A0,B0,F0,F1,REQ0);
cycle_apart.context(pco(A0,F,F0) & pco(B0,F,F1) &
                    mra(A0,REQ,REQ0) & mra(B0,REQ,REQ0));
cycle_apart.require(pco(F0,F,F1) | pco(F1,F,F0));
cycle_apart.error("A and B should be one cycle apart");
```

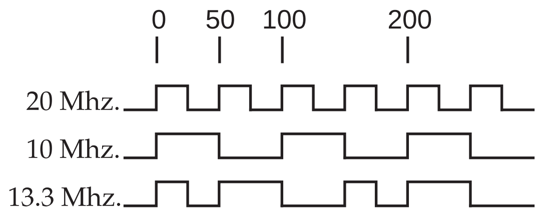
Figure 3.7: A sequential logic constraint requiring two events (A and B) to be one cycle apart. The context of the constraint establishes the identities of the quantified events: $F0$ is the clock edge prior to $A0$; $F1$ is the clock edge prior to $B0$; and $A0$ and $B0$ share the same REQ ancestor. The requirement is that $F0$ be the clock edge prior to $F1$ or that $F1$ be the clock edge prior to $F0$.

period be a constant, and thus the circuit presumably could have a dynamically changing clock. An alternative requirement might be that the period not vary (e.g., the third 13.3 Mhz. waveform would be illegal) in which case the formal specification should be amended as shown in the second version.

3.3 An Illustrative Example

In this section, we consider a circuit with an input wire X that obeys the Ethernet protocol (see Figure 3.9). Since the protocol consists of both functional and temporal properties, we can specify the behavior of the input wire with a cyclic OEgraph (for describing functionality) and a set of timing constraints (for describing temporal behavior). Much of the complexity of the Ethernet protocol is due to electrical properties which we cannot specify; however, the protocol does serve as a good illustrative example.

One of the major points which we will demonstrate using this example is that there is no clear distinction between functional and temporal behavior. Although we restricted



```
/* version 1 */
oe_wire CK("CK");
oe_constraint clock_cons("clock_cons");
oe_restriction CK_HIGH(CK ^ HIGH);
/* used to identify C1 and C2 as events with
   signal = CK and signal value = HIGH */
discrete_name C1("C1",CK_HIGH);
discrete_name C2("C2",CK_HIGH);
setup.quantify(C1,C2);
setup.context(nco(C1,CK_HIGH,C2));
setup.require((timeOf(C2)-timeOf(C1) >= 50)&
              (timeOf(C2)-timeOf(C1) <= 100));


/* version 2 */
...
discrete_name C3("C3",CK_HIGH);
setup.quantify(C1,C2,C3);
setup.context(nco(C1,CK_HIGH,C2) & nco(C2,CK_HIGH,C3));
setup.require((timeOf(C2)-timeOf(C1) >= 50)&
              (timeOf(C2)-timeOf(C1) <= 100)&
  timeOf(C2)-timeOf(C1)==timeOf(C3)-timeOf(C2));
```

Figure 3.8: Three possible clock signals for a circuit that should be clocked at "10 - 20 Mhz." —the third waveform is a composite produced from an available 2-phase clock and may or may not be a valid clock waveform for the circuit. Two versions of a constraint that describes an acceptable clock signal are given; the first one includes the third waveform, the second one excludes it.

event-logic and weakened its expressive power, the ability to specify a constraint that only applies in a specific context allows statements about functionality to be specified. For example, a constraint that only applies when the wire X is not high:

$$\ldots \wedge \neg\, X \textit{ is } \textbf{high} \Rightarrow \textit{timing requirement}$$

can easily be restated:

$$\ldots \Rightarrow X \textit{ is } \textbf{high} \vee \textit{ timing requirement}$$

The classification of behavior into functional and temporal aspects is thus certainly a loose one, and it should not be surprising that the Ethernet protocol can be specified in

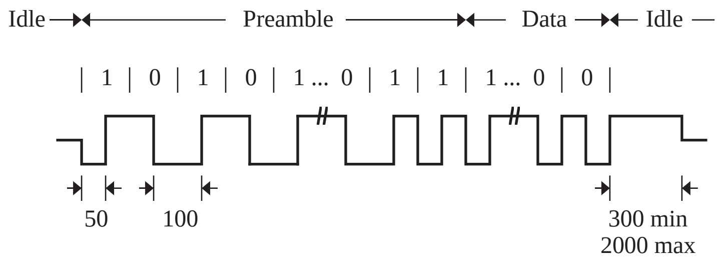


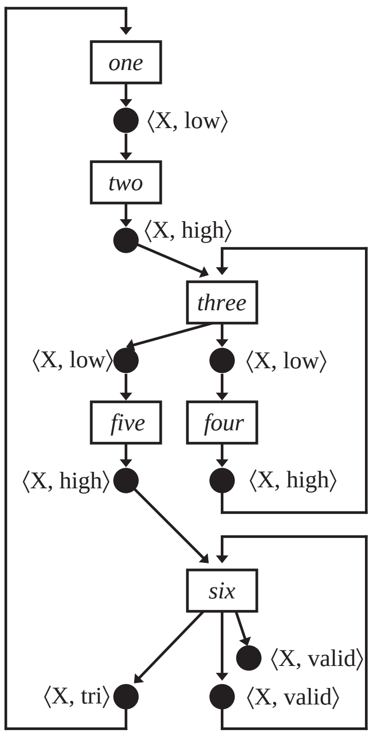
Figure 3.9: The Ethernet protocol for data transmission. The protocol consists of a preamble (which allows receivers to synchronize to the clocks of the sender) followed by data. Manchester encoding is used for both the data and the preamble (which is a fixed 64 bit alternating sequence of 1s and 0s terminating with 11). The transition from idle (tri-state) is to low and the transition to idle is from high (and must occur within 300 – 2000 ns.).

many different ways using OEgraphs.

The OEgraph on the left in Figure 3.10 contains six operations which completely characterize the Ethernet protocol. Operations *one* and *two* produce the first two transitions of the preamble, and operation *three* is on a loop with operation *four* that generates the alternating ones and zeros of the preamble. The preamble terminates with operation *five* and the data is generated by operation *six*.

The OEgraph on the right in Figure 3.10 simply states that some number of valid transitions will occur on X before a tri-state transition occurs and the behavior repeats. All of the temporal information (and a great deal of the functionality) is specified via timing constraints (the OEgraph is needed because the restricted constraint semantics cannot specify that events *must* occur). This is accomplished using unspecified values (see Section 3.1.2) which may appear in places other than propagation delays (e.g., unspecified iteration counts for loops). There are seven timing constraints which essentially state that X obeys the Ethernet protocol. The complete formal specification for both examples can be found in the Appendix A.

Now consider a hypothetical protocol which is very similar to the Ethernet protocol except that the preamble can be of variable width. Let us also assume that the data should be of bounded length. This constraint is easy to specify if the preamble



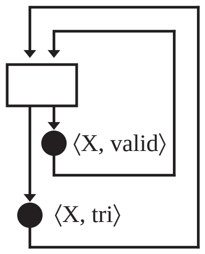


Figure 3.10: Two different operation event graphs for the wire X obeying the Ethernet protocol. Complete details for both specifications appear in the appendix.

is bounded, we would simply constrain the total separation time between the tri-state to low and high to tri-state transitions. However, if the preamble is not bounded, we need to specify a maximum separation between the end of the preamble, and the high to tri-state transition. This constraint will be easy to specify for the OEgraph on the left because there is an event (the output of operation *five*) corresponding to the last transition of the preamble (the event's name does not appear in the figure).

For the OEgraph on the right, we cannot refer to the event by name, but must use the expressive power of the representation to specify the constraint. We can quantify and, via the context, identify four consecutive transitions characteristic of the "11" at the end of the preamble (e.g., rising, falling, rising, falling each separated by 50ns). We also quantify and identify the tri-state-to-low transition and the next high-to-tri-state transition that surround the "11" transitions. The requirement would then state an upper bound on the time between the last rising transition of the preamble and the high-to-tri-state transition at the end of the data.

This constraint does in fact specify an upper bound on the length of the data. However, it also specifies an upper bound for any "11" that occurs in the data portion—see Figure 3.11. This is a good example of a useful constraint that actually constrains more events than necessary. Of course, if the distance from the first "11" to the end of the data has an upper bound, then specifying the same bound for other "11"s to the end of the data is perfectly acceptable even if it is redundant. If we wanted to specify a lower bound on the amount of data, however, we would not be able to do so. The constraint would erroneously insist that no "11" transitions occur near the end of the data (i.e., it is no longer redundant, and in fact creates a problem).

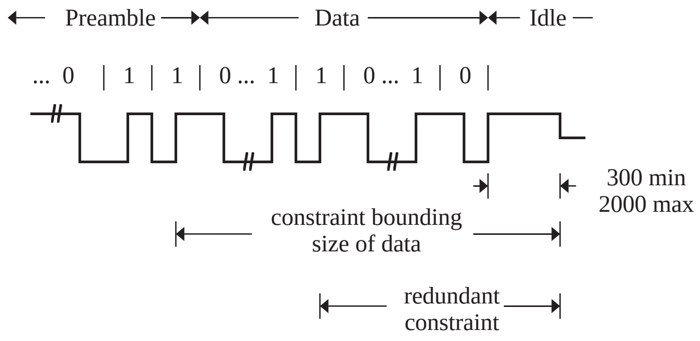


Figure 3.11: Hypothetical protocol with variable length preamble and a timing constraint stating that the data portion has a maximum length. The constraint is from "11" to the high to tri-state transition, and thus a redundant constraint from a "11" in the data would also exist.

The constraint cannot be formally specified (with respect to our simple OEgraph and the restricted event-logic) because the first "11" on X (after a transition from tri-state) cannot be identified. This example demonstrates why introducing hierarchy into the model is desirable. We could identify the "11" sequence of events as a hierarchical event and then trivially specify the constraint using the *nco* relation. At present, hierarchy is not supported, and the OEgraph would need to be modified so that the last event on the preamble was an event node in the specification.

The ability to describe behavior using both an operational and a denotational specification language allows designers to work at convenient point somewhere between the alternative approaches described by this example. In some cases both styles are valuable. For example, all of the timing constraints on the wire X in the second specification could also be attached to the first specification. Each specification would then serve as a check on the other.

Chapter 4

Validation

Specification languages allow designers to specify systems which can later be synthesized or analyzed by a variety of design automation tools. One critical problem, is to ensure that the specification captures the system that the designer wanted to specify. If the specification is inherently incorrect, the resulting formal analysis or implementation will be flawed with respect to the goals of the designer. Moreover, fixing problems of this nature can often be very time consuming because the relationship between the specification and the design tool's unexpected/undesired results can be difficult to understand (a process akin to debugging optimized code).

In order to remedy this problem, designers need to be able to visualize and explore their specifications; gaining an understanding of their specification as a separate and distinct entity. This process is referred to as *design validation* — specifications are validated with respect to the designer's informal conceptualization of the design. As discussed in Section 2.1.3, the ability to simulate a design (execute the specification) constitutes a very powerful form of design validation. The user emulates the system's environment and observes the specification's responses.

Many different benefits arise from the ability to execute an abstract specification. Users can correct errors in their specification and observe its operation under a number of different input scenarios. In some cases, the specification may be correct, in that it captures the intended system, but simulation may uncover unknown and unexpected patterns of behavior that are indicative of conceptual errors in the design itself. Typically, these errors would otherwise be discovered only after the design has been implemented, or, at the very least, after a great deal of time and energy has been expended. Specifications can be executed at various points in their development, and thus simulation also helps facilitate the process of specification, wherein the design is first crudely specified and then iteratively refined. One additional benefit is that the individuals that intend to use the design (the “customers”) can interact with the designer and the design early on, and benefit from the extensive *prototyping* that simulation provides. Executable specification languages have recently become quite popular, because the benefits of simulation have become quite apparent. Readers may want to consult [Harel 92] for a more in-depth discussion.

4.1 OEsim: A Simulator for OEgraphs

OEsim is a compiled simulator that takes as input an OEgraph (our textual specification language based on operation-event graphs) and generates an executable program linked to a simulation front-end. Users run this program, and step through the execution of the OEgraph: operations respond to input events by generating output events and changing internal state variables. Events generated by the simulation incur propagation delays that are randomly sampled from the specified upper and lower delay bounds for each cause statement (usually a uniform random distribution is used, but any distribution function can be specified). This allows the simulator to model hardware and other devices whose temporal behavior is not completely fixed (i.e., propagation delays often vary based on temperature changes, power fluctuations, etc.). The simulator contains commands which support user control (e.g., single stepping, setting breakpoints, scheduling events, etc.) and access to internal data structures (e.g., ancestry information, time of occurrence, etc.). The simulator is implemented in C++ and consists of approximately 2000 lines of code. By virtue of being a compiled C++ program, an operation node can include arbitrary C++ code that can be used to provide special interactions with the user (e.g., read and write data files, update customized windows or graphics, etc.). The functional simulation relies on standard discrete event simulation techniques.

A novel and interesting aspect of OEsim is its ability to report timing constraint violations. A violation indicates that the operational specification (the operation-event graph) describes an execution which the denotational specification (i.e., timing constraints described using the restricted event-logic) does not allow. Violations are found by observing the temporal and causal relationships that arise between the input and output events during a specific execution of the specification. The reporting of constraint violations often allows a user to detect design or specification flaws, or it may indicate that the user has incorrectly emulated the environment (e.g., a constraint from one input event to another is violated). Figure 4.1 contains a sample simulation that contains a constraint violation.

```
Welcome To Simulation v1.3, Mon Nov  5 15:13:43 1990
oesim-0> schedule-event F 0
oesim-0> schedule-wire D HIGH 60
oesim-0> schedule-wire D LOW 120
oesim-0> run-to 150
event_occurs at time:   0 event F
event_occurs at time:   25 event R
event_occurs at time:   50 event F
event_occurs at time:   60 event D$<external>(D*HIGH)
event_occurs at time:   75 event R
event_occurs at time:   100 event F
event_occurs at time:   120 event D$<external>(D*LOW)
event_occurs at time:   125 event R

 ** Constraint Violation: setup:
 r0 d0 : nco(d0, R, r0) ==> ((t(r0) - t(d0)) >10)
 r0 =  unique event: R occurrence: 3 at time: 125
 d0 =  unique event: D$<external> occurrence: 2 at time: 120

 stopped at time: 150

oesim-120>
```

Figure 4.1: A sample simulation of the flipflop in Figure 3.3, including a violation of the setup constraint of Figure 3.6.

In simulation, new discrete events occur and a specific execution (e.g., whose temporal properties may be based on randomly chosen delay values) is incrementally constructed. Each execution is, of course, potentially infinite, and timing constraints specify relationships with respect to this potentially infinite set. When constraints (expressed using event-logic) universally quantify discrete events, violations can be reported when a particular instantiation of discrete events causes a constraint violation. Thus, violations can be detected by analyzing a subset of the execution (e.g., a simulation in progress). With existential quantification, violations often cannot be reported because the events that satisfy the constraint may not yet have occurred. We could report the satisfaction of (and then dismiss) constraints with existentially quantified events but in simulation (i.e., for validation purposes) constraint violations are of primary interest.

We believe that constraint violations should be reported as soon as they can be detected. Users should not have to decide when to analyze a partial execution for constraint violations, nor should they be required to analyze a large "simulation dump" that contains a list of constraint violations and a trace of the model's execution. In order to understand a constraint violation, users may need to access internal data structures and issue simulation queries. Thus, *post mortem* analysis is often either difficult or expensive (because a large amount of data must be stored) and we believe that users often find it to be both awkward and wasteful as a constraint violation that occurs early during simulation may render the results of further simulation meaningless.

OEsim thus supports interactive simulation and reports constraint violations as soon as they occur. Unless directed otherwise, OEsim analyzes each partial execution for constraint violations. One of the motivations for our restrictions with respect to timing constraint specification in OEgraphs (see Section 3.2.2) was to enable the efficient analysis of constraint violations during simulation. Without our restrictions, analyzing event-logic formulas after each new discrete event occurrence would have been extremely inefficient. Our restrictions allow OEsim to perform incremental constraint checking in a more efficient manner without in practice weakening the expressiveness of our specification language for the purposes of timing constraint specification.

4.2 Incremental Constraint Checking

Incremental constraint checking is a mechanism that allows constraint violations to be reported as soon as they occur. The process is incremental, in that the timing constraints are examined each time a new discrete event occurs, yet each constraint is only checked once for each possible assignment of unique discrete events[1]. When a new discrete event occurs, constraint violations are detected using a three step process:

1. Based on the characteristics of the newly occurring event, identify the constraints which could possibly be violated. For each identified constraint,

2. Check to see if there are other previously occurring discrete events which, together with the newly occurring event, establish and satisfy the context of the constraint. For each set of discrete events that satisfy the context,

3. Check to see if the requirement for the constraint is violated.

For example, in a simulation of Figure 4.2, let us assume that X events had previously occurred at time 5 and 12 and that a new Y event occurred at time 17. The first constraint ("example1") would need to be checked because it universally quantifies discrete events which are occurrences of Y (i.e., the *discrete name* *Y0*'s restriction is the event Y). The second constraint ("example2") only quantifies events that are high transitions on the wire W and would thus not need to be checked. We then instantiate *Y0* to "Y at 17" and evaluate the context with $X0$ instantiated to "X at 5" and also "X at 12" and, since in neither case is the context true, no constraint violation would be reported. If a new X event then occurred at time 20, we would need to check the constraint again with *X0* instantiated to "X at 20" and *Y0* instantiated to "Y at 17" and, since the context

[1]Technically, if a constraint has more than one quantified discrete event associated with the same restriction, OEsim will evaluate the constraint multiple times when one or more of the quantified discrete events both instantiate the newly occurring event. In this case, detecting and removing this slight duplication of effort would be more costly than simply repeating the analysis.

```
oe_wire W("W");
oe_event X("X");
oe_event Y("Y");
...
discrete_name X0("X0",X);
discrete_name Y0("Y0",Y);
discrete_name WH0("WH0",W ^ HIGH);
discrete_name WH1("WH1",W ^ HIGH);
...
oe_constraint example1("example1");
example1.quantify(X0,Y0);
example1.context(timeOf(X0) > timeOf(Y0));
...
oe_constraint example2("example2");
example2.quantify(WH0, WH1);
...
```

Figure 4.2: An OEgraph fragment containing two timing constraints

is true, we would report a constraint violation if the requirement was false. We now examine each of these steps in more detail and present several important optimizations which improve efficiency.

4.2.1 Identifying which Constraints to Check

Whenever a new discrete event d occurs, the first step in checking for a constraint violation is to determine a set of constraint/discrete name pairs which need to be further examined. For each unique restriction r in the OEgraph, if $id(d, r)$ is true, then we need to examine each constraint containing one or more discrete names associated with r. For our example, if a high transition on wire W occurs, then we need to check the second constraint with *WH0* instantiated to the new event, and with *WH1* instantiated to previously occurring high transitions on W (and vice versa, with *WH1* instantiated to the new event, and *WH0* instantiated to previously occurring events).

If a constraint's context strictly orders the time of occurrence of two quantified events (e.g., the context requires $\tau(Y0) < \tau(X0)$) then we can avoid checking the constraint when the earlier event occurs, because we know that the context will be false (the later event has not yet occurred). In our example, we can thus ignore new occurrences of Y because we know *a priori* that there are no instantiations of discrete events which can satisfy the context. This static optimization is easily implemented, we analyze each constraint before running the simulation and detect which discrete names quantify events that must occur before other quantified events if the context is to be true. Note that we can apply the optimization to constraints involving ancestry relationships (mra) because an ancestor must always occur before its descendant.

The optimization can also be applied to constraints involving the chronological relations (pco and nco) although in this case our checking methodology must be slightly altered. These relations can be true if the two discrete events occur at the same time. Thus, if the context of our first constraint was "$nco(Y0, X, X0)$" we would know that any $X0$ satisfying this context must occur no earlier than $Y0$. We would like to evaluate the constraint only after occurrences of X, but we need to ensure that every $Y0$ that could satisfy the context has occurred. Thus, we delay the second step of our constraint evaluation until all of the discrete events that will occur at a particular time stamp have occurred. This is a slight deviation from our principle of reporting constraint violations as soon as they occur, but in practice the optimization improves efficiency and has little adverse impact because it is uncommon for more than a few events to occur at the same time.

4.2.2 Instantiating Discrete Events and Evaluating the Context

Conceptually, during simulation, we incrementally construct $\mathcal{X}_p$, a partial execution containing every discrete event that has occurred. For each unique restriction r, we incrementally construct the set of discrete events that are *instances* of r:

$$instances(r) = \{d \mid d \in \mathcal{X}_p \ \wedge \ id(d, r)\}$$

When a new discrete event d occurs, the first constraint checking step (described above) provides a constraint/discrete name pair. The event d will be instantiated into the given discrete name, and in the second step of constraint evaluation, we determine if the context can be satisfied by instantiating previously occurring discrete events into the other discrete names specified in the constraint. Thus, we need to examine each element of $instances(r)$ for every r corresponding to one of the other discrete names. All combinations of instantiations are tried, and if the context is ever true, the requirement is checked.

One problem with this approach is that the analysis takes an exponential amount of time with respect to the number of discrete names that appear in each constraint. For example, a constraint quantifying n occurrences of an event X will require $n(|instances(X)|^{n-1})$ context evaluations whenever a new X occurs. For example, if a constraint quantifies *X0*, *X1*, and *X2* and a new X occurs at time 50 with previous occurrences at times 10, 20, 30, and 40, the constraint would require 25 evaluations with *X0* instantiated to "X at 50" and

X1	*X2*
X at 10	*X at 10*
X at 10	*X at 20*
X at 10	*X at 30*
X at 10	*X at 40*
X at 10	*X at 50*
X at 20	*X at 10*
.	
.	
.	
X at 50	*X at 50*

A total of 75 new evaluations will result because we also need to evaluate the constraint with *X1* instantiated to "X at 50" and with *X2* instantiated to "X at 50".

An optimization akin to lazy evaluation can easily be used to improve efficiency. We evaluate the context without fully instantiating each discrete name, i.e., in our example, we could evaluate the context with *X0* instantiated to "X at 50" and with *X1* instantiated to "X at 10" and with *X2* uninstantiated. If the evaluation of the context returns **false** we can proceed by instantiating *X1* to "X at 20", etc. — otherwise (the evaluation returns **true** or **maybe**), we instantiate *X2* to "X at 10" and proceed. The choice of which discrete names to instantiate can be made either statically (e.g., in the order they appeared when quantified by the user) or by examining each constraint and using heuristics to establish a good order (e.g., if the context for our example was "$nco(X0, X, X2) \wedge nco(X2, X, X1)$" our choice of instantiation order, e.g., $X0, X1, X2$, would be a poor one, since *X0* and *X1* are not directly related to one another in the context).

4.2.3 Other Optimizations

Other more sophisticated approaches could be implemented. One obvious method would be to attempt to find which sets of discrete events satisfy the context. We are given a constraint/discrete name pair and thus know one discrete name which is instantiated with only one event (i.e., the given discrete name is instantiated to the newly occurring event d). We could then attempt to find the other discrete event(s) that must be related to this event if the context is to be satisfied. For example, if the context contains a *pco* relationship involving d, we could attempt to find the discrete event(s) that is the previous chronological occurrence, and avoid the computationally expensive task of stepping through every discrete event that is in $instances(r)$). Unfortunately, because the context is a Boolean formula, this approach is essentially the Boolean satisfiability problem (SAT) for which there is no efficient solution (SAT is NP-complete). However, in many cases, the context may simply be a series of conjuncted relationships, and specialized strategies for these cases could result in efficiency improvements.

The optimizations we have described help reduce the amount of time required for constraint checking. However, as the simulation progresses and new events occur, constraint checking becomes more time consuming because more events need to be instantiated each time a constraint's context is evaluated. This problem is related to another efficiency concern: the amount of space required to store past events in $instances(r)$ and maintain the directed acyclic ancestry graph. This is particularly troublesome in that larger simulations usually generate many events, and some events (e.g., clock edges) occur very frequently. However, it should be pointed out that larger simulations (i.e., large OE-graphs) are not inherently less efficient to simulate. Timing constraints apply to a small number of events, irrespective of graph size, and the efficiency of the simulation engine is related to the amount of parallelism inherent in the graph, not the graph size.

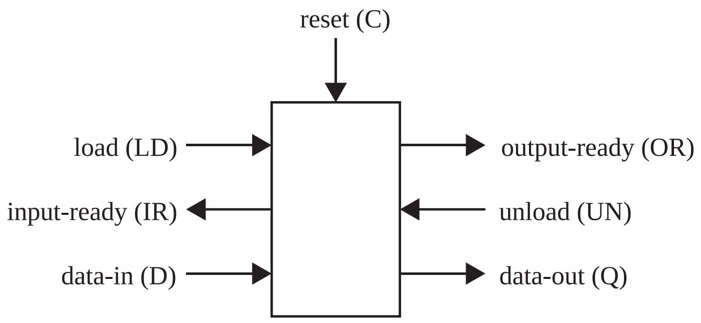
One approach to the problem of the simulation progressively slowing down due to constraint checking involves discrete event removal. Discrete events can be removed from $instances(r)$ if it can be shown that they are no longer needed for constraint checking. Before removing the discrete event, the ancestry information contained in the discrete event needs to be pushed outward. Note that we can trivially avoid storing any event d for which there does not exist a restriction r such that $id(d, r)$ is **true**. Many constraints involve simple chronological relationships that do not require storing complete histories (e.g. instead of storing every clock edge, only the two most recent edges are stored since they are the only ones involved in constraints). Thus, it might be possible to determine *a priori* how many events need to be stored. With respect to causality, often only the more recent causal chains are important, and in this case an event removal technique akin to dynamic garbage collection could operate periodically and effectively prune the ancestry graph. Optimizations of this nature have not been implemented in OEsim, and need to be further explored.

4.3 An Illustrative Example

In this section, we present an example to illustrate the use of OEsim. We assume the role of a designer needing to integrate a storage queue into a design. There are certainly a large number of commercial ICs we could choose from, and in this example we will use the Texas Instruments SN74LS222 FIFO. Ideally, a specification for the chip would already exist in a library of OEgraph specifications, and we would simply integrate it into our design and begin prototyping. However, if the specification is not in the library, we will need to specify the chip before we can explore whether or not it is a suitable choice for our design.

The ability to work at a high level of abstraction is quite desirable. We have no knowledge of exactly how the SN74LS222 is implemented, and we wish to specify its behavior as quickly and as succinctly as possible. Our primary source of information will be the databook specification, part of which appears in Figure 4.3. The databook spec-

Figure 4.3: A portion of the databook specification for the SN74LS222 16 element FIFO storage buffer (used by permission, © 1986 Texas Instruments, all rights reserved).



Timing Constraints

	to	MIN
LD↑	LD↓	60
LD↓	LD↑	15
UN↓	UN↑	30
UN↑	UN↓	30
LD↓	UN↓	50
UN↑	LD↑	50
Dv	LD↓	50
C↓	C↑	20

Figure 4.4: The LS222 uses a 4 phase handshake and requires minimum separation times for correct operation

ification is informal, and it is quite possible that errors will arise when the specification is formalized via OEgraphs. OEsim can be used to detect these errors and help iteratively refine the specification, until its behavior (during simulation) corresponds with the expected behavior of the LS222. A complete OEgraph specification for the LS222 appears in the Appendix (see B.1), the input and output signals are shown in Figure 4.4. Our specification captures both functional and temporal properties. There are several minimum delay separation constraints which must be respected if the circuit is to work properly, and estimates for the minimum and maximum propagation delays for the circuit are obtained from the databook (our specification makes use of a parameterized timing constraint subroutine, "pulse_min," which is used to simplify the specification of the timing constraints).

We use modularity to simplify the process of creating an additional instantiation of the LS222 for our design. Our specification, unlike the examples in Chapter 3, defines a new operation *class* (a new library element), which can be instantiated into a design. Both the timing constraints and the dependency arcs are encapsulated within the specification of the LS222 library component. For example, the LS222 only holds 16 data

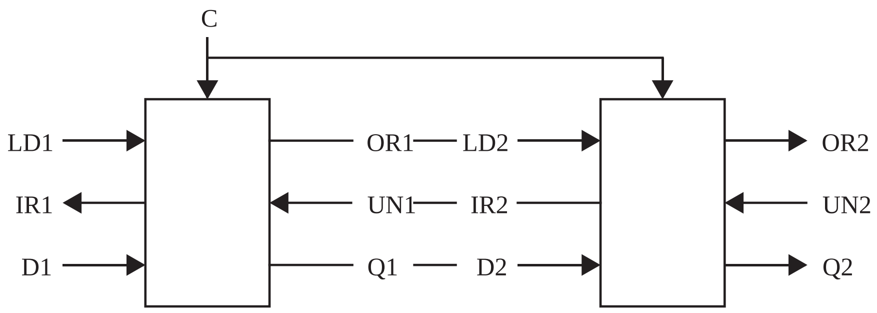


Figure 4.5: Two LS222 chips connected together.

elements, but is composable — we can obtain a 32 element FIFO by joining two of the chips together (see Figure 4.5). The complete specification appears in Figure 4.6 — we simply instantiate two instances of the LS222.

```
include "runtime.h"
include "LS222.h"
oe_wire not_CLR("not_CLR");
oe_wire LD1("LD1");
oe_wire D1("D1");
oe_wire IR1("IR1");
oe_wire Q1_D2("Q1_D2");
oe_wire OR1_LD2("OR1_LD2");
oe_wire UN1_IR2("UN1_IR2");
oe_wire UN2("UN2");
oe_wire OR2("OR2");
oe_wire Q2("Q2");

main(int argc, char* argv[]) {

oe_operation_sn74ls222 fifo1("fifo1",
not_CLR, LD1, UN1_IR2, IR1, OR1_LD2, D1, Q1_D2);

oe_operation_sn74ls222 fifo2("fifo2",
not_CLR, OR1_LD2, UN2, UN1_IR2, OR2, Q1_D2, Q2);

oesim(argv[0], options(argc,argv));
}
```

Figure 4.6: The OEgraph specification for two LS222 chips connected together. Note that the specification simply instantiates two copies of the LS222 chip (see Appendix B.1). The timing constraints are encapsulated within the specification of the LS222 library component.

A simulation log demonstrating the use of the LS222 (e.g., containing requests to load and unload the queue) appears in the Appendix, see B.2. The simulation reports a timing constraint violation: the data being moved from the first chip to the second (signal

D2_Q1) is not valid 50 nanoseconds before the signal to load the second chip is issued (OR1_LD2 is lowered). This constraint violation indicates that a simple composition of the two chips will not always work correctly. Indeed, the databook specification shows a schematic diagram of two chips being interconnected and includes a 10 nanosecond delay element inserted between the two interior signals (OR1 and LD2) which apparently solves this problem. We can easily insert this delay element into our specification and observe that the simulation does not report any constraint violations.

This, of course, does not guarantee correct operation of our new design. We would need to exhaustively simulate the circuit for every possible propagation delay using a comprehensive set of environmental stimulii (e.g., choosing load and unload requests that fully exercise the circuit). Alternatively, we could use formal verification techniques (the subject of the next two chapters) to detect the original problem and verify that the proposed solution is correct. This is precisely the approach taken by [Martello et al. 90] from which we have taken this example. We should point out, however, that verification is not a substitute for simulation. Simulation allows us to validate our specification and to understand how to use the LS222 in our design. One other interesting result of simulation is the discovery that the timing constraints for the composed circuit are not identical to that of a single LS222. For example, in the original circuit, a request to unload the FIFO must occur at least 50 nanoseconds after a load request. This constraint must be satisfied for each LS222 in the composed circuit, and thus an unload request (to the second FIFO) occurring 75 nanoseconds after a load request (to the first FIFO) may lead to a timing constraint violation, because the load (of the first FIFO) may generate an internal request to load the second FIFO. This request incurs a propagation delay with respect to the original load, and thus the two signals (the load and unload request for the second FIFO) can be too close together. Thus, although the circuit will work correctly (if the load and unload requests are sufficiently separated), the timing constraints do not compose modularly as one might assume.

4.4 Discussion

We have used OEsim to describe a wide range of examples derived from real circuits or extracted from the specification and synthesis literature — the largest being the specification and simulation of the Intel Multibus (see [Amon et al. 91]) for which we were able to include all of the constraints described in the databook specification, many of which can not be expressed using more restrictive event-based specification languages (such as the event-graphs of [Borriello 88b]). At the University of California at Berkeley, OEsim has been used to represent and simulate the abstract interfaces of complex components that are interconnected on a printed circuit board or multi-chip module (see [Sun & Brodersen 92]). A modified form of our restricted event-logic has also been incorporated (see [Ortega 92]) into the heterogenous simulation framework of Ptolemy [Buck et al. 91] to support the specification and checking of timing constraints when working within its event-driven simulation domain.

We have not analyzed the performance of OEsim in detail, but have found it to be efficient and capable. Most timing constraints require a separation time between two events, and thus many timing constraints contain only two quantifications because a simple chronological or causal relationship is sufficient to identify the events being constrained. When specifications contain mostly events (and not wires) it is often the case that only two events will need to be quantified since the constrained events can be referred to by name. When timing constraints apply only to some transitions on signal wires, timing constraint specification is often more difficult since the events must be identified via complex contexts (which often quantify additional events). This point was illustrated by the Ethernet example in Chapter 3 which also demonstrates (see constraint "c2" in Appendix A.2) that, in some cases, a large number of simple constraints can be equivalently expressed as a single more complicated constraint (e.g., 63 constraints that quantify 3 events vs. one constraint that quantifies 4 events).

OEgraphs were designed with simulation in mind. A clear simulation semantics was a requirement for all the features of the model. An important goal consisted of modeling general timing constraints expressed using both chronological and causal relationships. The challenge was ensuring that the calculus used for timing constraint specification had a clear simulation semantics and would support incremental constraint checking. This was accomplished by restricting the event-logic and adding the three new primitives (mra, nco, and pco). OEsim was developed to help explore the operation-event graph specification paradigm, and faithfully adheres to the underlying semantics of OEgraphs. For example, when multiple events occur at the same time, OEsim randomly chooses which event to remove first from the event calendar in the simulation engine.

OEsim, as illustrated by our example, supports both modularity and encapsulation and constitutes a useful design automation tool for prototyping, especially when temporal relationships must be specified and taken into account. A novel feature is its support for both operational and denotational specifications, and its ability to check for consistency between the two specifications during simulation. Timing constraints need to be formally specified, and OEgraphs provide a more structured and natural methodology as opposed to the current ad-hoc approach of simply hand-coding an operational specification to look for and report constraint violations. It is unlikely that OEsim or a derivative will soon be adopted by the design community (due to the popularity of VHDL, Verilog, and other existing HDLs for which simulators have been built). However, the inherent strengths of the operation-event graph paradigm and the practicality of validating such specifications using simulation have been demonstrated.

Chapter 5

Verification

Verification is a procedure that compares two formal specifications. In some cases, one specification represents a design and the other specifies properties that need to be verified. Frequently, the specifications represent designs at two different levels of abstraction, i.e., a design specification is compared to a design implementation. Verification techniques are then used to establish that the implementation satisfies the requirements defined by the specification. For example, in combinational logic verification (see [Bryant 86]), the Boolean function that specifies the functionality of a piece of combination logic is compared with the gate level implementation (see Figure 5.1).

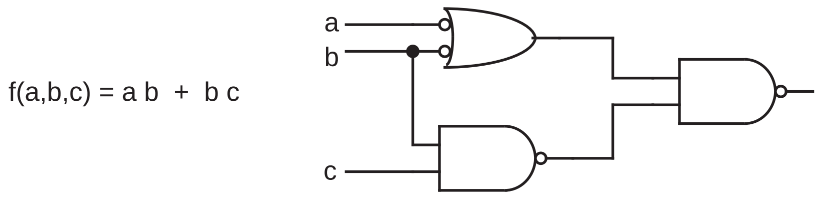


Figure 5.1: Verification is a formal procedure in which two specifications are compared. For example, we could formally prove that the combinational logic on the right implements the functionality of the Boolean equation on the left.

Verification is inherently a more difficult problem to solve than validation. In validation, a specific instance of behavior exhibited by the specification is generated. In verification, all possible behaviors of the specification must be taken into account. For example, in Figure 5.1 we could observe that the Boolean function and the circuit implementation have the same output value (zero) when all of the inputs are assigned to zero. In order to formally verify that the circuit and function are equivalent, we need to check and make sure that the outputs of both are identical for every possible combination of input values. Unfortunately, even for simple verification problems like this one (e.g., there is a discrete set of input values and we have ignored temporal issues) exhaustive simulation is not computationally feasible because of the large number of computations which must be performed (i.e., given n inputs there are 2^n possible input assignments).

In this remainder of this chapter, we examine three very different methodologies that can be used to help automate the process of design verification: *symbolic simulation*, *interactive proof managers*, and *exploration of state space*. We then discuss and examine interface timing verification, the focus of the subsequent two chapters.

5.1 Symbolic Simulation

One general technique for performing verification is based on exhaustive validation (simulation). Symbolic variables are used to represent multiple input values or ranges of operating conditions. In one simulation, a symbolic simulator can compute results that would require many simulation runs using a conventional simulator because each input would need to be completely specified.

This technique is quite adequate for verifying combinational logic circuits [Bryant 86]. A Boolean function is obtained from the gate-level implementation via symbolic simulation, and verification is successful if the resulting function is equivalent to that of the specification. If propagation delays of the gates are taken into account, hazards can be detected using a symbolic simulation technique in which the propagation delays are considered to be symbolic variables (see [Ishiura et al. 89]). A related technique involves the use of an *extended value system*, in which special values are used to indicate unknown or indeterminate values (e.g., the value X is used to indicate inputs which are either zero or one). Verification tools based on this methodology can disregard aspects of the design which are not important (e.g., in some cases data values can be ignored if only temporal correctness is being verified). This approach has been used to improve the efficiency of verification techniques for synchronous systems (see [Bryant & Seger 91]).

Due to the inherent complexity of verification, some verification tools are *incomplete*, in that they verify a design's specification against an incomplete set of all possible behaviors of the implementation. *Incomplete verifiers* formally compare the results of simulation with the design specification. If some of the inputs (such as data) are not fully specified, and some support for symbolic simulation is included, these tools can analyze a large set of system behaviors.

Most incomplete verifiers (e.g., SCALD [McWilliams 80], and TDS [Kara et al. 88]) provide little support for specifying and verifying temporal properties other than simple setup and hold constraints. One incomplete verifier that supports a more expressive specification language is CLOVER [Doukas & LaPaugh 91], which uses an event-based specification language (ATCSL) that relies on event indexing to identify the discrete events being constrained.

OEsim could be classified as a verifier because it compares two different specifications: the operational specification described by the operation-event graph, and the denotational specification described using event-logic. It is an incomplete verifier, because it only examines a very small subset of all possible behaviors (namely, the one begin simulated). In fact, most simulation and analysis tools could be classified as incomplete verifiers, and the use of the term 'verification' in this context is somewhat misleading. Incomplete verifiers cannot be used to formally argue for the correctness of a design because the complete set of system behaviors is never examined or taken into account.

5.2 Interactive Proof Managers

Another misleading term: *theorem provers*, is used to describe a general class of verification tools that establish the correctness of a design through the use of formal theorems and proofs. Ostensibly, theorem provers are given a theorem and then automatically determine whether or not this theorem is true (e.g., they prove that "`implementation` $\Rightarrow$ `specification`", where `implementation` and `specification` are formal mathematical statements describing the specification and implementation of the design).

In practice, theorem provers are essentially proof managers that help users manage the complexities of a proof which is constructed by hand. They are highly interactive, and help users decompose large proofs into smaller manageable components. Once users are familiar with the formal syntax and semantics that the prover supports, they are able to easily construct new proofs (at the same level of abstraction) using techniques learned from experience, as well as templates and tactics that were used in previous proofs. Learning to use a theorem prover requires a significant investment of time and energy. Therefore, most individuals use only one theorem prover. Although the formal differences between the various provers are in many cases well understood, there is little practical difference between them. Theorem provers used for circuit verification include HOL [Gordon 88], LP [Garland & Guttag 89], LAMBDA [Fourman 90], and Boyer-Moore (see [Hunt 85]).

The primary advantage of this approach to verification is the flexibility and expressiveness of the specification paradigm. Temporal issues can be fully taken into account, and complex circuits can be verified through the use of modularity and abstraction. The primary disadvantage, of course, is that the techniques are not fully automated, and require a large investment on the part of the user.

5.3 Exploration of the State Space

One way to examine the space of all possible system behaviors is to explicitly generate a description of the state space and then examine this description in order to verify properties of the system. For example, reachability analysis can be used to determine whether or not anomalous behavior can arise (e.g., "Can signal x ever go high without y having previously gone high?"). This methodology has been used for a variety of verification tools. The major limitation is that the state exploration algorithms become computationally infeasible as the number of states becomes very large.

One well known approach, *model checking*, verifies properties expressed using temporal logic. For example, [Burch et al. 90] use the temporal logic CTL [Dill & Clarke 85] to verify properties of sequential circuits. Recent work in this area relies on symbolic state encoding using Binary Decision Diagrams (BDDs) [Bryant 86]. This allows systems with large numbers of states to be verified (e.g., systems with 10^{20} or in some cases even 10^{120} states [Burch et al. 91]). Model checking has typically been used to verify systems in which propagation delays between circuit elements do not vary and are assumed to be unit delays.

Many verification problems can be expressed and solved using algorithms for language containment. In this approach, a formal language is used to express both the specification and the implementation. Trace theory has been used quite succesfully to verify speed independent asynchronous circuits [Dill 88], and in [Burch 92], timed asynchronous circuits are verified using extensions to the verifier of [Dill 88]. Finite automata are used to specify the sets of acceptable traces, and the verification algorithms are also state based.

Another approach to the verification of systems with variable propagation delays is based on the idea of *bifurcation*. This technique creates a state transition graph, but only creates new states when the operation of the system will change based on different choices for propagation delays. Examples of this approach are [Devadas et al. 92] and [Martello & Levitan 93].

5.4 Timing Verification

Timing verification is able to provide guarantees about the temporal behavior of a system and is thus very useful in design synthesis, because these guarantees can be used to help guide the process of transforming a design from an abstract specification into a realizable implementation. Implementations need to meet a given standard of performance (e.g., operate fast enough) while minimizing implementation costs (e.g., power consumption, area, etc.). Guarantees about timing behavior are quite valuable because designs usually have a great deal of freedom with respect to temporal behavior. Timing information can help ensure that resources are only allocated where they are needed. If performance constraints can be met without additional resources, they should not be unnecessarily wasted.

Many design tools have been created to help analyze the performance of a design with respect to its timing behavior. For example, tools exist to help determine the rate at which a circuit can be clocked, or if the setup and hold time constraints will be satisfied. Although these tools can be considered verifiers, they are usually referred to as *timing analyzers*. Most work at low levels of abstraction, e.g., on combinational or sequential logic circuits. Because of the very limited semantics of both the system and the behavior that is verified, analytical techniques which do not require state-based exploration or symbolic simulation often suffice. Some of these tools require human assistance (e.g., in CRYSTAL [Ousterhout 85] users need to identify the direction of current flow for some transistors) but most are fully automated. Many interesting problems with respect to timing analysis have been explored (e.g., the false-path problem, see [McGeer & Brayton 91]) and we will not attempt to further categorize work in this area.

Our interests with respect to timing analysis are in the area of *interface verification*, which relies on event-based abstractions as opposed to gates and transistors. The events of the system being verified (e.g., an interface between a processor and a memory chip) correspond to system activities and logic transitions (e.g., clock edges, time intervals of valid addresses on busses, positioning of read and write strobes, etc.). Interface verification algorithms attempt to determine minimum and maximum separation times between system events given the propagation delays of the specified system (e.g., to verify that data will appear on the bus shortly after a read request). An event graph (see Section 2.3) is used to capture the events and propagation delays of the system.

Shortest path algorithms have been used to solve problems that contain only linear constraints, e.g., [Borriello 88b] and [Brzozowski et al. 91]. Both [McMillan & Dill 92] and [Vanbekbergen et al. 92] provide an overview of the various types of event graphs and timing constraints for which verification algorithms have been developed (both authors present algorithms for handling non-linear constraints).

Most of the analysis in this area has dealt with acyclic event graphs, and unfortunately even apparently simple classes of verification problems are NP-complete [McMillan & Dill 92]. In [Martello et al. 90] a cyclic system representing the LS222 of Section 4.3 is verified, but the verification technique is of limited applicability due to a restricted semantics that requires systems to be specified using a very simple event-graph.

In the next two chapters, we explore two different interface verification techniques based on the event-based specification paradigm. In Chapter 6 we present a verification tool for a restricted class of operation-event graphs and event-logic. The most interesting aspect of this work is that the verifier supports the use of symbolic variables for representing propagation delays and timing constraints. In Chapter 7 we present a fundamental timing analysis problem that is a prerequisite to extending the work of Chapter 6 to handle more complex concurrent systems.

Chapter 6

Symbolic Timing Verification

Symbolic timing verification is a powerful extension to traditional constraint checking in that it allows propagation delays to be bounded by variables as well as fixed values. The verifier then determines actual allowable delay bounds — constraints on the values of the variables, taking into account the interface, throughput, and latency constraints (see Figure 6.1). Verification of this sort is valuable for timing-directed synthesis algorithms because it provides a clear picture of how changes to a specification affect both performance and constraint satisfaction. If propagation delays are fully specified, but the timing constraints are expressed using symbolic variables, verification can be thought of as an analysis procedure because it will determine the constraints on performance that can be satisfied.

In this chapter, we present an approach to symbolic timing verification using constraint logic programming techniques. Our verification tool handles a small subset of the OEgraph specification language presented in Chapter 3. Unfortunately, formal timing verification is a difficult problem, and the expressiveness of OEgraphs makes verifying a general specification intractable. By restricting the functionality (i.e., via a fixed set of operations) and by restricting the timing constraint syntax and semantics, timing verification is feasible.

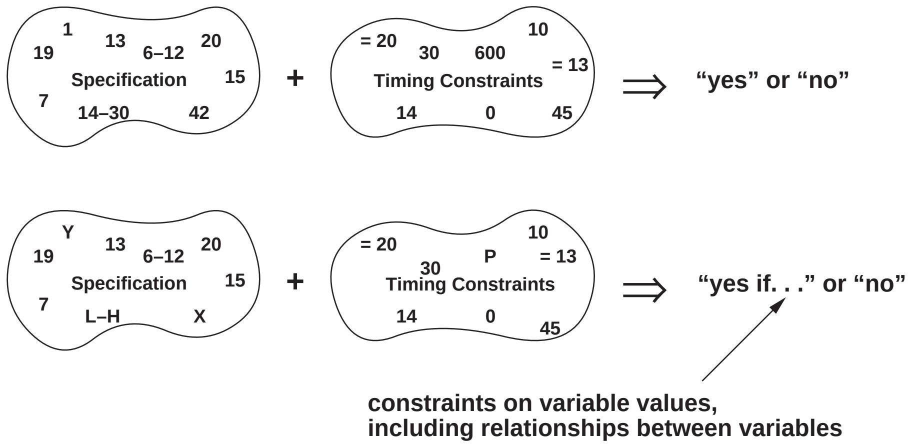


Figure 6.1: Timing verification traditionally (top) provides a "yes" or "no" answer to the question: "Will every execution of the specification always satisfy the timing constraints?" If symbolic variables are used for propagation delays and/or timing constraints (bottom), verification will be dependent upon the value of these variables, and might be successful only if some of these variables are constrained.

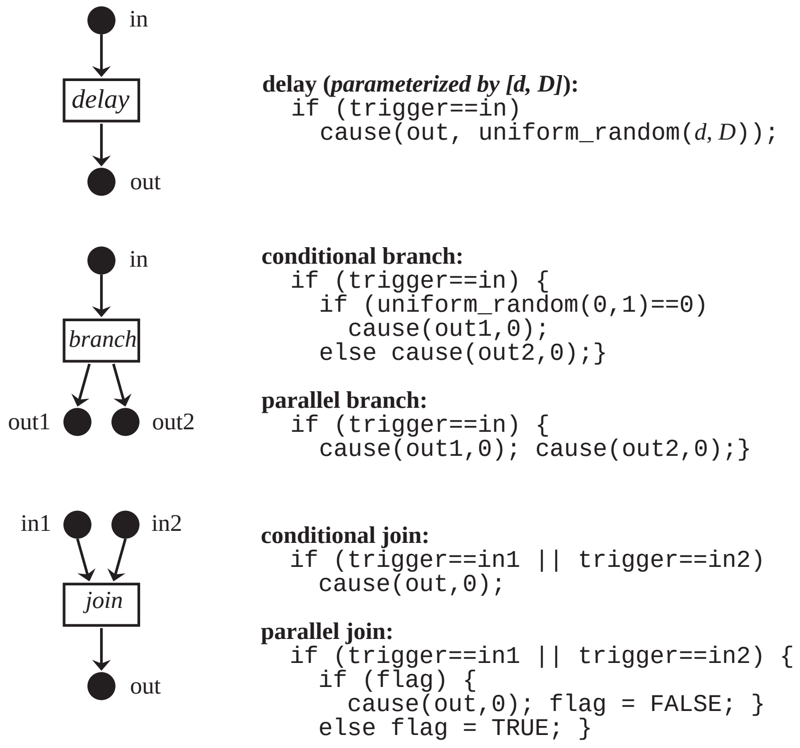
6.1 Verification Model

We model our behavioral specifications using a bipartite directed graph whose two node types are *events* and *operations*. Events serve as reference points for timing relationships, and operations encapsulate different ways to interconnect events. Events can represent changes in logic level on circuit wires (from a structural perspective) or data and control flow in a hardware description language (from a behavioral perspective). Operations represent circuit functionality by taking events as inputs and determining which event outputs to generate. For the purposes of timing verification, we use a subset of operation-event graphs consisting of five basic types of operations and each type has a fixed indegree/outdegree and a simple semantics (see Figure 6.2).

delay operations model the passage of time. Each occurrence of the output event is delayed by some real number in the closed interval $[d, D]$, where d and D are either integers or integer variables, constrained such that $0 < d \leq D$.

parallel-branch and *parallel-join* operations are used to model the initiation and completion of parallel activity. A branch immediately causes both output events to occur; a join waits for both input events before immediately causing the output.

conditional-branch and *conditional-join* operations are used to model non-deterministic choice, a branch immediately causes one of the two output events to occur after the input event occurs; a join waits for either input to occur before immediately causing the output event.



delay (*parameterized by [d, D]*):
```
  if (trigger==in)
    cause(out, uniform_random(d, D));
```

conditional branch:
```
  if (trigger==in) {
    if (uniform_random(0,1)==0)
      cause(out1,0);
    else cause(out2,0);}
```

parallel branch:
```
  if (trigger==in) {
    cause(out1,0); cause(out2,0);}
```

conditional join:
```
  if (trigger==in1 || trigger==in2)
    cause(out,0);
```

parallel join:
```
  if (trigger==in1 || trigger==in2) {
    if (flag) {
      cause(out,0); flag = FALSE; }
    else flag = TRUE; }
```

Figure 6.2: Our verification model consists of five basic types of operation nodes. A fragment of the OEgraph operation code for each operation is shown.

The events (E) and the operations (O) are connected in a bipartite graph (G) to form a single process (the cyclic graph). All events have an indegree and an outdegree of one, and the graph is strongly connected. A process is a useful construct that restricts control flow by disallowing multiple activations of an operation, i.e., when an input to a delay operation occurs, another input cannot arrive before the output occurs. We also require the graph to be *series-parallel*, i.e., every branch has a matching join of the same type (conditional or parallel), and there is an event in the graph such that on every forward path back to the event, if a join is visited, then its matching branch must have already been visited (see Figure 6.3).

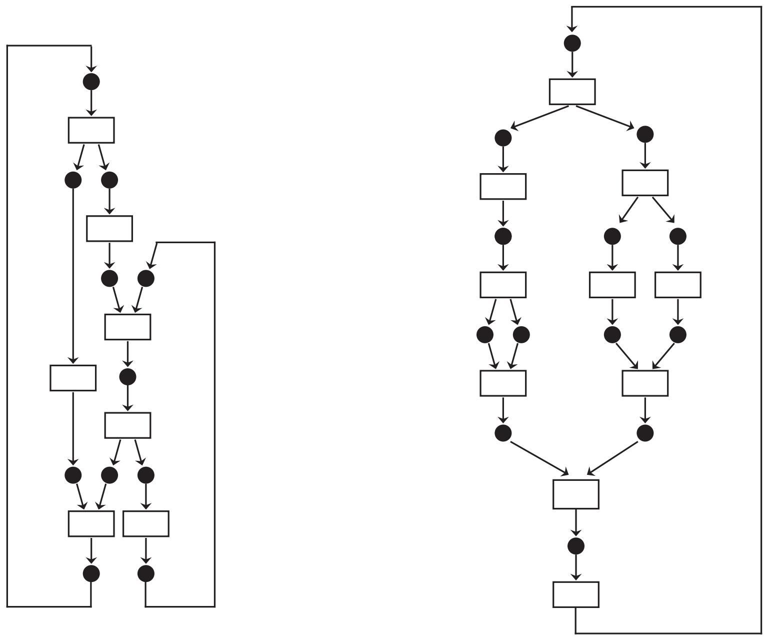


Figure 6.3: The graph on the left is not series-parallel. Our verification model requires graphs to be series parallel, like the graph (process) on the right.

6.1.1 Timing Constraints

Timing constraints specify minimum and maximum separation times between events in the graph, and are expressed using a very restricted form of the event-logic notation introduced in Section 3.2.2. Formally, a constraint is a 5-tuple: $\langle \mathit{from}, \mathit{to}, \mathit{type}, \mathit{comp}, \mathit{sep} \rangle$, where:

from and *to* are events and elements of E,

type is used to specify which discrete occurrences of *from* and *to* are being constrained, i.e., $\mathit{type} \in \{\mathit{mra}, \mathit{nco}, \mathit{pco}\}$, *comp* specifies the separation time comparison operator, i.e., $comp \in \{\leq, \geq, =\}$,

sep is either an integer or an integer variable.

The formal semantics of each constraint is defined using an event-logic formula: [1]

$$\forall f\ \forall t\ \ \mathit{Identify} \Rightarrow\ \ (\mathit{Context}\ \Rightarrow\ \mathit{Requirement})$$

where

$$\begin{aligned}
\mathit{Identify} &= (\mathit{event}(f) = \mathit{from})\ \wedge\ (\mathit{event}(t) = \mathit{to}) \\
\mathit{Context} &= \begin{cases} \mathit{nco}(f, \mathit{to}, t) & \text{if } \mathit{type} = \mathit{nco} \\ \mathit{pco}(t, \mathit{from}, f) & \text{if } \mathit{type} = \mathit{pco} \\ \mathit{mra}(t, \mathit{from}, f) & \text{if } \mathit{type} = \mathit{mra} \end{cases} \\
\mathit{Requirement} &= (\tau(\mathit{to}) - \tau(\mathit{from})\ \mathit{comp}\ \mathit{sep})
\end{aligned}$$

6.2 Verification Methodology

Given a graph G and a set of constraints we need a verification methodology capable of answering the fundamental question: "Will all of the constraints be satisfied during every execution of G?" Furthermore, the verification technique must operate symbolically so that the symbolic variables used in specifying propagation delays and required separation times (i.e., d, D, and *sep*) are taken into account.

One obvious approach to verification would be to examine each constraint and determine whether or not it will be satisfied during any execution of G. The presence of symbolic variables could result in an inequality (involving some of the variables) that must be met if verification is to be successful. If there is no feasible solution to the entire set of inequalities generated from all of the constraints, verification would fail. Unfortunately, the inequalities may be non-linear (due to the presence of minimum and

[1]This is a very small subset of the restricted logic discussed in Section 3.2.2. Only two discrete events appear in each constraint, they are related via a simple *nco*, *pco*, or *mra* relationship, and the requirement specifies a minimum or maximum separation time between occurrences of the two events.

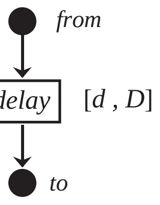
maximum functions introduced by the conditional and parallel branches) and thus determining whether or not there is a feasible solution is non-trivial. The methodology we present in this chapter combines these two steps (generating inequalities and looking for a feasible solution) into a single integrated approach.

We use a constraint transformation technique which reduces timing constraints to other simpler constraints and possibly one or more additional linear inequalities. The transformation strategy essentially consists of a list of rules which specify how to decompose constraints. The methodology is best understood via an example.

6.2.1 Delay Operations

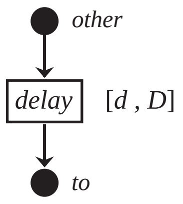
Only two basic transformation strategies (rules) are used for constraints with a *to* event (recall that constraints are of the form $\langle \mathit{from}, \mathit{to}, \mathit{type}, \mathit{comp}, \mathit{sep}\rangle$) that is the output of a *delay* operation. The strategies do not need to take the constraint *type* into account because the different semantics are identical with respect to constraints spanning delay operations. The strategies do vary based on the constraint comparison. We present the rules used for a maximum separation time constraint, i.e., given a constraint $\langle \mathit{from}, \mathit{to}, \mathit{type}, \leq, X\rangle$

1. If there is a delay operation with input *from* and output *to*, i.e.:



   then transform the constraint into the inequality $D \leq X$.

2. Else if there is a delay operation with input *other* and output *to*, i.e.:



then transform the constraint into the inequality $D + T_1 \leq X$ and the constraint $\langle \mathit{from}, \mathit{other}, \mathit{type}, \leq, T_1 \rangle$ (the variable T_1 is a new symbolic variable created as a result of this transformation).

Now consider a graph consisting only of delay operations, e.g., the graph in Figure 6.4, and the constraint $\langle a, c, \mathit{type}, \leq, 20 \rangle$. The second rule will be applied first yielding the inequality: $D_{b,c} + T_1 \leq 20$, and the new constraint: $\langle a, b, \mathit{type}, \leq, T_1 \rangle$. This new constraint is transformed by the first rule into the inequality: $D_{a,b} \leq T_1$. We can then report a symbolic result by removing internal variable T_1 (i.e., $D_{a,b} \leq T_1 \leq 20 - D_{b,c}$) yielding: $D_{a,b} + D_{b,c} \leq 20$. Of course if these delays are specified as integers and not as symbolic variables, we can report whether or not this inequality is true (and thus wether verification is successful).

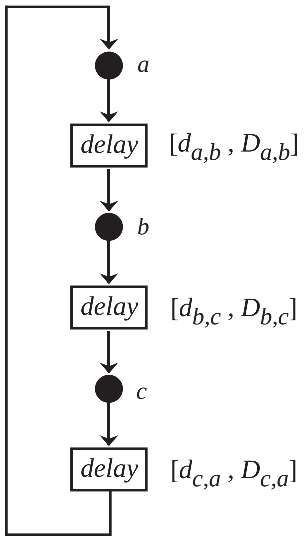


Figure 6.4: A graph consisting of three delay operations

6.2.2 Implementation

Our symbolic timing verifier based on this methodology has been implemented using a constraint logic programming language. See [Cohen 90] for a comprehensive introduction and overview of this programming paradigm. We chose the language CLP(R) because it provides an ideal framework for exploring this methodology. CLP(R) incorporates an incremental version of the standard simplex linear programming algorithm and a goal based programming language akin to Prolog. Its use has enabled us to focus attention on the verification rules instead of the mechanics of the symbolic linear programming, unification, and search/backtracking aspects of the methodology. It is possible that a more efficient version of the verifier could be constructed but the version of CLP(R) that we have used [Jaffar et al. 92] has proven to be quite adequate.

The verification tool takes as input a description of the process and a list of constraints to be verified. An initialization step constructs data structures that will be used by some transformation rules (e.g., information from static graph analysis) and adds default constraints requiring all delay range minimums to be greater than zero, and all delay range maximums to be greater than or equal to their respective minimums (i.e., $0 < d \leq D$).

The verifier currently consists of over 2500 lines of CLP(R) code implementing well over one hundred transformation rules. A large number of rules are needed to account for the quality of the verification tool (i.e., its ability to detect and correctly process complex behaviors), the different types of operations (and the resultant behaviors which they allow), and the different types of constraints and their semantics. Much of the complexity is due to extensions to the basic operations which will be discussed in Section 6.4. Some of the complexities arise due to the formal semantics of our constraints, others arise due to the inherent complexity of the constraint with respect to the functional specification (e.g., a constraint from an event in one side of a parallel branch to an event on the other side).

6.2.3 Transformation Rules for Join and Branch Operations

In this section, we present additional rules which demonstrate some of the features of our approach and a few of the subtle issues that the rules must take into account.

In Chapter 3 we argued that both chronological and causal relationships are needed to support timing constraint expression. Most of the rules are dependent upon the type of relationship used to identify the discrete events being constrained. Thus, most of the rules apply only for constraints of a specific type (*nco*, *pco*, or *mra*). The easiest type to deal with is the *mra* because it constrains pairs of events based on ancestry (causal) relationships, and thus the constraints correspond directly to paths in the graph. The chronological relationships (*nco* and *pco*) require, in general, more sophisticated reasoning.

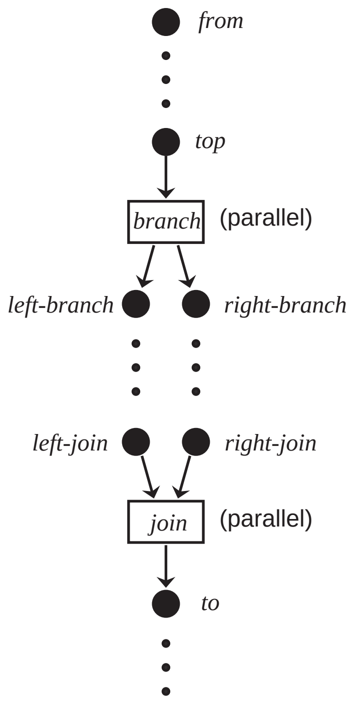
These two types are used to specify how constraints are interpreted in execution sequences such as: a, a, a, b, b, b (i.e., three discrete occurrences of a followed by three discrete occurrences of b). An *nco* constraint from a to b would constrain every a with the next chronological occurrence of b, i.e., the time of the first occurrence of b is important and the times of the second and third occurrences of b are irrelevant. A *pco* constraint from a to b would constraint the previous chronological occurrence of an a with respect to every occurrence of b, i.e., the time of the last (third) occurrence of a is important and the times of the first and second a's are not relevant.

The rules for a conditional branch/join are quite dependent upon whether the constraint is an *nco* or a *pco*. Recall that the conditional branch is nondeterministic, and therefore it is unknown to the verifier which of the two branches will be taken for a particular pass through the branch. This makes it impossible to verify a constraint $\langle \mathit{from}, \mathit{to}, \mathit{nco}, \leq, \mathit{sep} \rangle$ when *from* is outside the branch/join and *to* is inside on either the left or right side of the conditional operation. The constraint requires a maximum separation time between every *from* and the next chronologically occurring *to*, but there is no guarantee that when the conditional branch's input event occurs that it will lead to the event *to* (i.e., the other side of the branch could be chosen). Thus, there is no upper bound on the separation time (a minimum time separation/lower bound can easily be obtained). A similar rule applies to *pco* constraints when *from* is inside the conditional branch/join and *to* is outside.

At present, our verification methodology is restricted to handling non-deterministic conditional branches. Ideally our model should make use of additional information regarding when and how often a branch is taken in order to properly verify more complex systems. However, even if these extensions are implemented, there is clearly a difference between the *nco* and *pco* relationships that must be taken into account.

We now consider transformation strategies for constraints whose *to* event is the output of a parallel join operation. Two different transformation rules are used, depending upon whether or not the *from* event is inside or outside the parallel branch.

Given a constraint $\langle \mathit{from}, \mathit{to}, \mathit{nco}, \geq, X\rangle$ if there is a parallel join operation with output *to*, and if *from* is outside of (and not the input event to) the matching parallel branch, i.e.,



then (assuming the other events are named as in the diagram) transform the constraint $\langle \mathit{from}, \mathit{to}, \mathit{nco}, \geq, X \rangle$ into:

1. the constraint $\langle \mathit{from}, \mathit{top}, \mathit{nco}, \geq, T_1 \rangle$
2. and either:

   the new inequality $T_1 + T_2 \geq X$, and
   the new constraint $\langle \textit{left-branch}, \textit{left-join}, \mathit{nco}, \geq, T_2 \rangle$.
3. or alternatively:

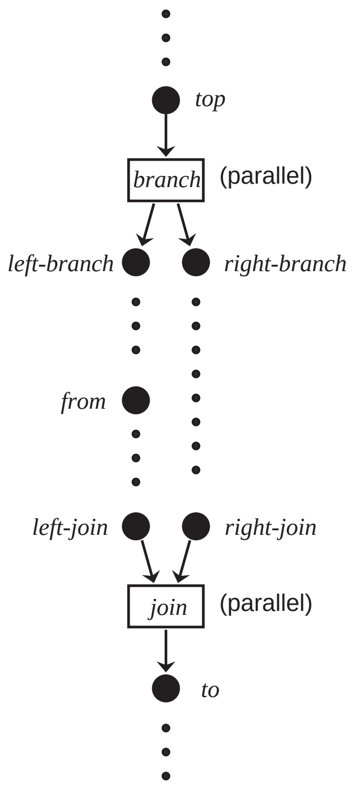
   the new inequality $T_1 + T_3 \geq X$, and
   the new constraint $\langle \textit{right-branch}, \textit{right-join}, \mathit{nco}, \geq, T_3 \rangle$.

Since *to* will not occur until both *right-join* and *left-join* occur, a minimum separation time may be ensured even if only one path is sufficiently slow. This represents an implicit choice for the verifier, which must determine which path to constrain. If verification is successful (i.e., the path can be slowed down without adversely affecting the verification of other constraints) it need not constrain or examine the other path. If verification fails (i.e., slowing the path down is not possible) the other path will need to be examined, and the verifier will need to backtrack.

The constraint could be expressed as an inequality involving a maximum function (e.g., the maximum of the two path delays must be sufficiently large). The verification problem is thus inherently nonlinear, although it can be formulated as a potentially exponential number of linear programming problems. Our methodology relies on a search paradigm — the search space being the many linear solutions to the non-linear formulation. Our approach is more efficient than the alternative generate and test methodology, which converts constraints into nonlinear inequalities and then solves each expansion of the nonlinear inequalities until a solution is found or all expansions fail. In our methodology, an incremental linear programming algorithm is used. This means that solutions which fail can be detected without requiring a full expansion of the nonlinear inequalities (i.e., failure can occur even if many of the constraints have not yet been transformed).

This early termination can prune large segments of the search space for which there is no solution. The incremental transformation process is also easy to understand (and extend) since it consists of a small number of simple rules used for different constraint semantics and different operations.

Returning to our constraint, we now consider the case where *from* is inside the branch (without loss of generality we will assume it is on the left branch), i.e.,



then (assuming the other events are named as in the diagram, and *from* is not equal to *left-branch* or *left-join*) the constraint $\langle \textit{from}, \textit{to}, \textit{nco}, \geq, X \rangle$ is transformed into:

1. the new constraints:

   $\langle \textit{right-branch}, \textit{right-join}, \textit{nco}, \leq, T_1 \rangle$,

   $\langle \textit{left-branch}, \textit{from}, \textit{pco}, \geq, T_2 \rangle$, and

   $\langle \textit{from}, \textit{left-join}, \textit{nco}, \geq, T_3 \rangle$

2. and either:

   the new inequality $T_1 \leq T_2 + T_3$, and

   the new constraint $\langle$*from*, *left-join*, *nco*, $\leq, X\rangle$.

3. or alternatively:

   not$(T_1 \leq T_2 + T_3)$, i.e., this inequality must fail, and

   the new inequality $T_4 + T_1 - (T_2 + T_3) \leq X$, and

   the new constraint $\langle$*from*, *right-join*, *nco*, $\leq, T_4\rangle$.

The rule for when *from* is inside the parallel branch uses two strategies: if *right-join* will always occur earlier than *left-join* (occurring via *from*) then the constraint may be transformed into a constraint from *from* to *left-join*, otherwise we must take into account how much later *right-join* could occur with respect to *left-join*. One interesting aspect of this rule is that the original *nco* constraint is transformed into a number of constraints, one of which is a *pco*. This is necessary since *from* could be inside an inner fork and the constraint $\langle$*left-branch*, *from*, *nco*, $\geq, T_2\rangle$ would then fail and lead to verification failure even though the original constraint can be sucessfully verified (because we are assuming that a *from* event occurred).

6.2.4 Zero Delay Semantics

Constraints that span operations with zero delay (e.g., branch and join) create additional problems for our verification methodology because of our choice of semantics. For example, consider the constraint: $\langle a, b, nco, \leq, 10\rangle$ with respect to the process graph of Figure 6.5. Although it would be natural to assume that this constraint is trivially satisfied because there is no delay in the parallel branch operation which generates a b after seeing an a, the constraint is in fact not satisfied and verification should fail. The problem is that our notion of "next chronological occurrence" is based solely on the times at which the discrete events occur. Since a and b occur at the same time, our formal definition of *nco* (see Section 3.2.2) concedes that the order of a and b cannot be disambiguated and thus the next b with respect to a might in fact be interpreted as a later occurring b. Since there is no guarantee that the next b will occur within 20 time units (because of the delay before event a) the constraint fails.

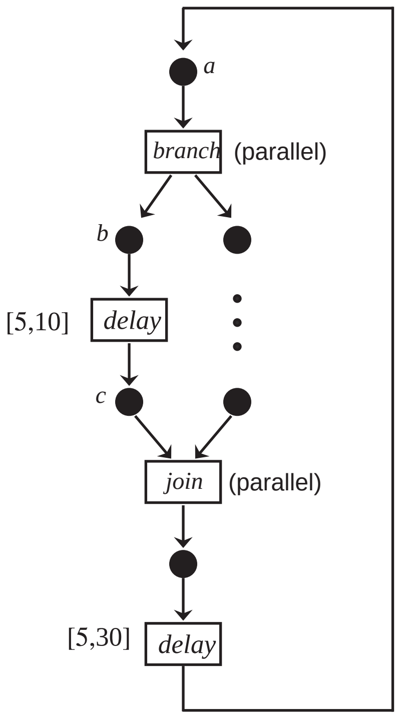


Figure 6.5: A simple process graph used to illustrate problems with regards to our formal semantics and operations with zero delay such as the parallel branch and the parallel join.

Although this result is unexpected, it is consistent with our semantics and our decision to base the semantics only upon the times of the event occurrences, and not on the notion that a caused b. Other semantics (e.g., *mra*) are used for expressing constraints between events that are causally related. A problem, however, develops with regard to the constraint $\langle a, c, nco, \leq, 20\rangle$ which is clearly satisfied. Our methodology will essentially transform this constraint into the constraint $\langle a, b, nco, \leq, 10\rangle$ which, as we have just discussed, fails.

The problem is that we have lost temporal information, namely that some delay between a and c exists, when we transformed the constraint from a to c into the constraint from a to b. One solution to this problem would be to require branch and join operations to also incur some non-zero delay and to incorporate this delay into our transformation rules. However, zero delay elements are very valuable because they support modularity (i.e., the ability to hook two components together) and abstraction. Many specification languages support the specification of zero delay elements. Zero delay semantics are often quite subtle (e.g., see [Ishiura et al. 90]) yet removing them can create other problems. For example, the language VHDL does not support a zero time delay. This has caused problems for translation schemes from domain specific application languages with zero time delay into VHDL [Vahid & Gajski 91].

Our solution to this problem is to create additional rules to handle constraints when the *from* event is an input to a zero delay operation. In these cases, the constraint is analyzed with respect to the graph topology to determine if a zero delay path to the *to* event exists. Based on this analysis, the constraint is then appropriately transformed.

6.3 Example

We now present results of symbolic timing verification to demonstrate the types of analysis which can be performed using the rules discussed above. Our verification tool is asked to verify a single constraint: $\langle g, a, nco, \leq, 30\rangle$ with respect to the process graphs shown in Figure 6.6.

Two different sets of propagation delays were used. The solution to the first set of propagation delays (shown on the left) is a linear inequality, $D_{g,h} + D_{j,k} + D_{m,a} \leq 30$, that closely resembles the constraint being proven, in that the delay operations along the path from g to a are constrained. This solution was obtained by detecting that k will always occur after l (when k occurs via g).

The second solution (for the graph on the right) is an inequality, $d_{b,c} \geq 20$, that specifies a minimum bound for the delay operation connecting b to c. This result is not intuitively obvious, because the delay operation from b to c is not on the path from g to a, and moreover in order to make g to a be fast, we must ensure that b to c is

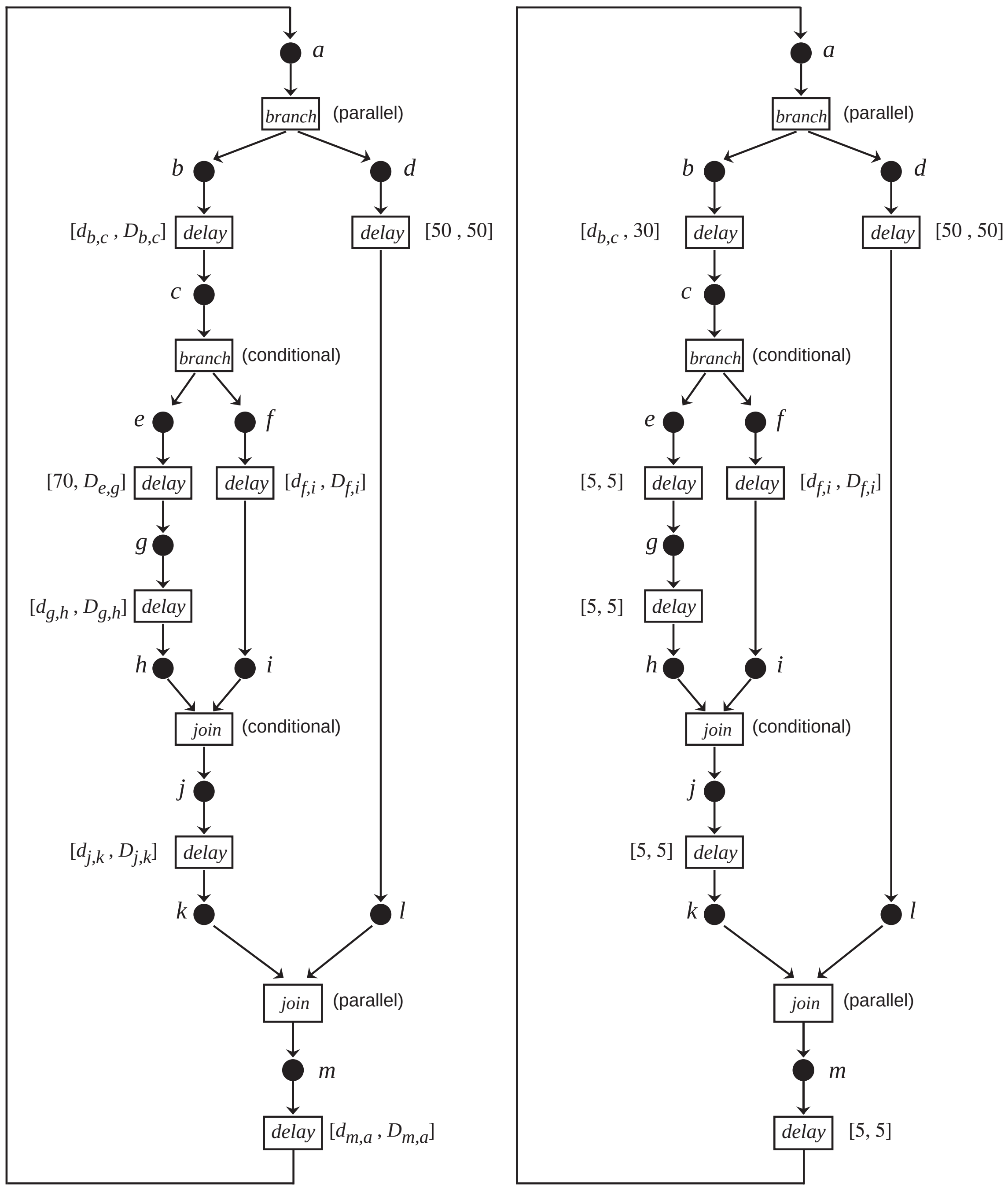


Figure 6.6: Two process graphs for which the verification tool analyzed the constraint $\langle g, a, nco, \leq, 30\rangle$. Verification was successful for both graphs, and yielded the linear inequalities: $D_{g,h} + D_{j,k} + D_{m,a} \leq 30$ (for the graph on the left) and: $d_{b,c} \geq 20$ (for the graph on the right, which is identical to the graph on the left except that the propagation delays are different).

sufficiently slow. The solution can, however, be easily explained. In this case, it is not possible to show that k will always occur after l (if it was, the constraint from g to a would be satisfied since the delay along that path adds up to 15 which is ≤ 30). The solution requires that g not occur too early with respect to l since k will end up waiting. This is accomplished by delaying event c (and thus delaying g). Note that some of the propagation delays specified as variables do not appear in the two solutions (e.g., the delay from f to i). This indicates that these delays are irrelevant with respect to the timing constraints being verified.

One benefit of performing symbolic timing verification that is not evident from this example is that our verifier can handle multiple constraints and multiple symbolic variables, and provide results that relate the constraints and the propagation delays with one another. These relationships (which in our tool must take the form of linear inequalities) can provide insight into the specification as well as provide valuable information for other design tools.

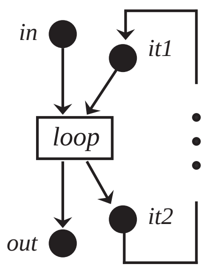
6.4 Extensions

In this section, we present three extensions to the verification model described in Section 6.2, and discuss both the strengths and weaknesses of our model.

6.4.1 Iterative behavior

One obvious limitation of the model is that it does not support the expression of iterative behavior. This is easily remedied by adding a new operation (depicted in Figure 6.7).

A *loop* operation is used to model iterative behavior and has two parameters that specify lower and upper bounds on the number of iterations that any given execution of the loop might execute. The lower bound is a non-negative integer, and the upper bound is either ∞ (i.e., there is no upper bound) or a positive integer greater than or equal to the lower bound.



loop (*parameterized by lower and upper bounds on the number of iterations*):

```
if (trigger==in) {
  count = 0;
  last = uniform_random(lower,upper);}
else if (trigger==it1)
  count = count + 1;

if (count < last)
  cause(it2,0);
else
  cause(out,0);
```

Figure 6.7: An operation that supports iterative behavior.

New transformation rules are needed in order to handle the *loop* operation. In some cases, the number of iterations need not be taken into account (e.g., *pco* and *nco* constraints that are always to or from the first or last occurrence of an event inside a loop) and in others the upper and lower bound provides information which is used with (multiplied by) the delay incurred during one iteration to determine lower and upper bounds on the total amount of time that can be spent in the loop. Incorporating the *loop* operation into our verification model was quite easy, and demonstrates the inherent flexibility of our approach, especially in this case, where a small number of additional rules are able to extend the expressiveness of the model.

One limitation we did not address is that symbolic variables cannot be used to constrain the number of iterations. Thus, it is not possible to determine via symbolic analysis the number of iterations that can be executed without violating any constraints. This information could be used by synthesis algorithms to determine, for example, how frequently data can be sampled without incurring any delay penalty (e.g., to improve the performance of one part of a design when the overall performance is constrained by another parallel activity). Unfortunately, the bounds on the number of iterations are multiplied by other symbolic variables (representing delay information from *it2* to *it1*) and if the bounds were expressed using symbolic variables then non-linear equations would result. At present, our symbolic verification methodology is limited to handling linear constraints, and thus specific integer bounds are required.

In fact, the solutions that we report are always linear constraints on the symbolic variables (or, of course, simply success or failure). For many problems, there may be multiple solutions to symbolic verification and we will report only a linear subset (e.g., we will never express a solution using a minimum or maximum function — instead, we will pick one of the two potentially feasible solutions). In theory, because our methodology is based on a search (i.e., Prolog), we can ask the verifier to backtrack and obtain other solutions. In practice, however, this is an additional limitation of our methodology, because some of our verification rules make use of the ability to *cut* the search space, and thus a retry will not always succeed in producing other alternative linear solutions.

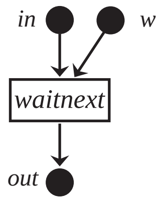
6.4.2 Additional Inequalities

Our constraint syntax and semantics is, as we observed, quite limited compared to the expressiveness of OEgraphs. One feature of our methodology that can sometimes be used advantageously to express more complicated constraints is the ability to specify linear inequalities with respect to the symbolic variables. These inequalities further constrain verification, which, in order to be successful, must find a feasible solution that also satisfies the additional constraints. These inequalities can be quite useful, for example, in relating two timing constraints to one another. A constraint stating that one event has to happen after another event can be specified using two *mra* constraints (when the events are causally related via a common ancestor) with symbolic separations and an inequality that constraints the two symbolic variables.

6.4.3 Communicating Processes

One serious limitation of the verification model is that it does not handle multiple processes. It is quite likely that from a behavioral perspective most concurrent systems are expressed not as a single process (with parallel activity) but rather as a set of communicating processes.

In this section we examine a simple extension that allows limited inter-process communication and is also useful for modeling clocked systems. The issues that arise from introducing this new operation (see Figure 6.8) are indicative of the complexities inherent in handling more general inter-process communication operations and cross-process constraints.



```
waitnext:
  if (trigger==in)
    active = TRUE;
  else if (trigger==w && active) {
    cause(out,0);
    active = FALSE;
  }
```

Figure 6.8: An operation that supports inter-process communication. The event w has no indegree and is an external event.

The *waitnext* operation blocks when the input event occurs and waits for the next occurrence of an external event before generating an output. Any external events (which can be any event in another process) occurring when the *waitnext* is not blocked are ignored. The semantics are similar to the Verilog "wait" statement.

This new operation allows processes to communicate and our verification methodology is augmented with new rules that allow constraints between events in different processes to be verified. New rules are also needed for constraints between events in the same process. For example, we present a rule used for propagating constraints across a *waitnext* operation:

Given a constraint $\langle \mathit{from}, \mathit{to}, \mathit{pco}, \geq, X \rangle$ if there is a waitnext operation with external input w and with output *to* and input *other* then (assuming *from* and *to* are both in the same process and *from* is not equal to *other*) transform the constraint into:

1. the new inequality $T_1 + T_2 \leq X$,
2. the new constraint $\langle \mathit{from}, \mathit{other}, \mathit{pco}, \leq, T_2 \rangle$, aand
3. the new constraint $\langle \mathit{other}, w, \mathit{nco}, \leq, T_1 \rangle$.

This rule results in a cross-process constraint being generated which essentially asks "How long might the operation wait for the external event?"

For the single process constraints described in the previous sections, the verification methodology is never pessimistic. The verifier never fails to report success if a feasible solution exists (this has not been formally proven, see Section 6.5). This is not the case for cross-process constraints where higher degrees of concurrency complicate the transformation rules. We have implemented a number of rules which serve to identify delay bounds which can result in successful verification. However, in many situations these bounds can be further tightened by noticing properties of the specification and the behavior of the circuit. It may be possible to analyze and detect these behaviors but some complex situations may prove quite difficult to verify. Thus our verification tool will be pessimistic, capable of verifying most constraints but occasionally failing to verify constraints which can in fact be satisfied.

Two simple rules demonstrate how bounds may be achieved, albeit pessimistically. Consider the cross process constraint $\langle other, w, nco, \leq, T_1 \rangle$. One simple way to determine an upper bound on T_1 is to assume that *other* could have occurred just after a w event and thus must wait for the process generating w to complete a full cycle of operation, i.e., we pessimistically transform the constraint into $\langle w, w, nco, \leq, T_1 \rangle$. This worst case strategy is useful in the case of operations waiting for clock edges. Frequently such operations have to wait an entire clock cycle before receiving the required edge, and the worst case strategy is appropriate. If we change the separation such that we need to establish a lower bound on T_1, i.e., for $\langle other, w, nco, \geq, T_1 \rangle$, we can trivially assume a pessimistic lower bound of zero.

If two processes communicate, it is likely that the points of interconnection between them can provide information to help tighten the pessimistic bounds described above. Consider the process fragment shown in Figure 6.9. The constraint $\langle a, g, nco, \leq, X \rangle$ can be verified by determining upper bounds on the amount of time that a and f might have to wait for a *tick* and bounding the delay from b to f. A naive approach would be to assume that both a and f have to wait an entire *tick* cycle. A better result can be obtained by noticing that the two *tick* inputs to the *waitnext* operations are related.

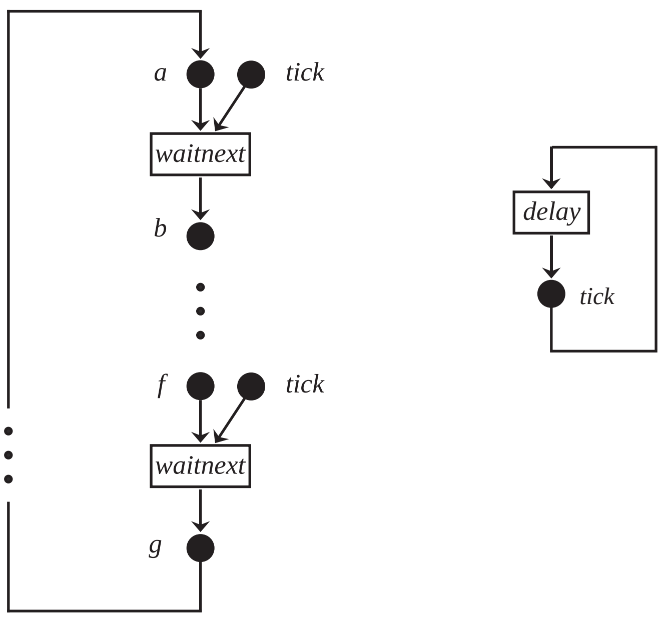


Figure 6.9: An example of two processes with one way communication

For example, if a *tick* is generated every 10 time units, and there is a delay of at least 5 (and less than 10) from b to f, then f will have to wait at most 5 for the next *tick* (not 10). Of course, uncertainty in both the delay between *tick* events and the delay between b and f may need to be taken into account, and this uncertainty if large enough could result in the pessimistic assumption being the best result.

In some situations, communicating processes adversely affect verification strategies by introducing circularities into the worst-case verification methodology. This can be seen in the process fragment shown in Figure 6.10. In order to calculate an upper bound on the delay from w to the next w we need to know the delay from x to the next a, which in the worst-case can be computed as the delay from a to a. This computation will require the delay from b to w which is pessimistically the delay from w to w, and we have a circularity. The two processes are mutually dependent, and the pessimistic strategy is not applicable (and in fact we must add rules to the verifier to detect and prevent circularities).

Although we have implemented various rules for dealing with a limited class of cross-

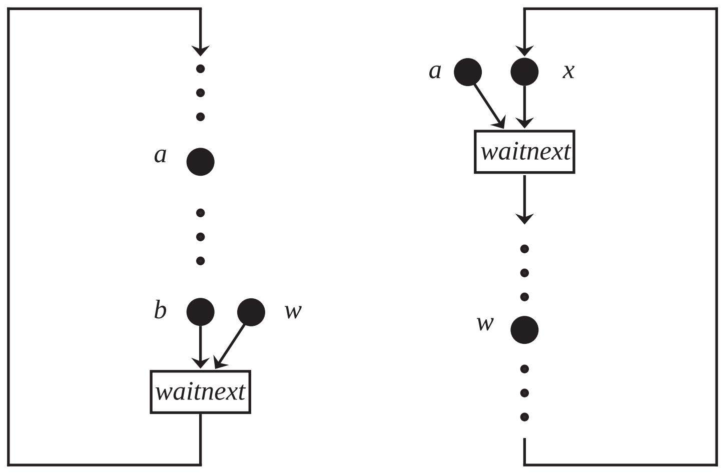


Figure 6.10: An example of two processes and interlocked communication

process constraints and inter-process communication, process synchronization introduces complexities that should be studied independently of the other aspects of our verification model (e.g., iterative constructs, conditional behavior, etc.). This is the subject of the next chapter.

6.5 Discussion

In summary, we believe that symbolic timing verification is a critical tool in the development of higher-level synthesis tools. The information obtainable from a symbolic verification process has three principal uses: implementation verification, obtaining constraints for synthesis, and design evaluation. Implementation verification confirms that an implementation of a design and its associated delays will meet the constraints in the original specification. Synthesis tools can use symbolic delay values to determine the degree of flexibility that is available while still satisfying the timing constraints. This can lead to much more efficient use of resources in the final implementation. Design evaluation can be performed by using symbolic values in the constraints and determining bounds on the values of these variables given circuit delays. This can provide information about how well a design will perform and also relate the constraint variable to circuit delays.

In this chapter we have presented a verification tool that uses a new programming paradigm, constraint logic programming, to perform symbolic verification for a restricted class of system behaviors and timing constraints. Although we believe that extensions to the model are clearly needed, it can be used to model a variety of concurrent systems and to verify a large class of temporal relationships. Our verifier has been used to analyze several real-world examples containing inter-process communication and cross-process constraints. The largest being a partial specification of the Intel Multibus which consisted of five communicating processes containing a total of 75 events, 65 operations, and 35 timing constraints (this example is discussed in [Amon & Borriello 92]).

The computational complexity of our methodology has not been studied in great detail. In the worst case, our approach is no more efficient than simply generating the non-linear set of inequalities for each constraint, and then exploring this space using an exponential search for a feasible linear solution. Most of the examples we have verified were quite trivial, and verification was quite efficient (e.g., the examples in Figure 6.6 took 4 seconds of CPU time on a Sparc II). We believe that execution speed will scale well to larger specifications due to the fact that most constraints describe relationships that are local and interrelate a small number of symbolic delays. Of course, some constraints can be quite complicated, but it is unlikely that larger specifications will contain many such complex interrelated constraints. In any case, the complexity of our approach is dominated not by graph size, but by the complexity of the interrelationships and how they affect the solution space. Verification of the Multibus took 2 minutes of CPU time on a Sparc II.

It should be emphasized that we have made little attempt to optimize execution speed. Some efficiency issues were addressed. For example, putting the new inequalities before the new constraints in every rule because this can prevent needless generation of constraints when the inequalities cannot be satisfied. Execution speed could be dramatically improved through the use of caching. At present, the results of graph analysis are not being re-used. This results in an excessive amount of execution time being devoted to repeating analyses that have already been performed, especially when the verifier is forced to backtrack.

One issue that we have not addressed is that of formally establishing the correctness of our verification rules. This would clearly need to be done if the verifier were to be used for anything other than exploring the feasibility of verification using our model and methodology. The correctness of the rules, at present, is based only upon a very careful analysis of the rules with respect to our constraint syntax and the types of behaviors allowed by the verification model. Some errors in the verification rules have been discovered through extensive testing, but it is of course quite possible that other errors exist.

In many ways, our methodology is similar to that which could have been employed using a theorem prover capable of performing efficient symbolic linear programming. With such a tool, each rule would constitute a theorem, and the transformation strategies would correspond to tactics for breaking up complex proofs (constraints) into less complicated ones. Our rules were created manually by taking into account both the semantics of our constraints and the possible behaviors of the model. However, the presence of synchronizations and cross-process constraints sufficiently complicates the model such that a deeper and more theoretical analysis of the core issues is needed.

Chapter 7

Analyzing Concurrent Systems

In this chapter, we consider a very restricted class of concurrent systems and examine a fundamental temporal analysis problem. Consider a simple concurrent system containing three processes that synchronize over two channels (a and b) and performs some internal computation. Such a system might be described textually using a concurrent programming language, as in the following figure:

```
Process 1 ::            Process 2 ::            Process 3 ::

repeat {                repeat {                repeat {
  Synchronize a;          Synchronize a;          Synchronize b;
  Compute [4, 10];        Compute [1, 2];         Compute [5, 20];
}                         Synchronize b;        }
                          Compute [1, 6];
                        }
```

Figure 7.1: A concurrent system synchronizing over two channels (a and b) with internal computation (delay ranges specified in brackets).

There are many questions regarding the temporal behavior of this system that we might ask: "How slowly might the first process cycle?", or "How long might the second process be idle waiting on the third process?", or "How can we best speed-up the performance of our system?" and "Which delays impact performance the most?" To answer such questions we need to be able to determine how the inter-process synchronizations affect the temporal behavior of our concurrent system. For example, the first process could obviously cycle with a period of at least 10 time units (the upper bound on the computation time) but how much more delay might be incurred as a result of the synchronization with the second process?

In this chapter, we derive an exact algorithm that determines tight upper and lower bounds on the separation in time of an arbitrary pair of *system events*, where an event can be thought of as an execution point in the system (i.e., the completion or initiation of computation or synchronization). Depending on the level of abstraction in the specification, events may represent low-level signal transitions at a circuit interface or may represent control flow in a more abstract behavioral view. We model a concurrent system (with no conditional or iterative behavior) as a cyclic connected graph. The nodes of the graph represent events and the arcs are annotated with lower and upper bounds on delays between events.

Our model permits the representation of both blocking and non-blocking communication. In the blocking case (used in Figure 7.1), two interacting processes wait for a synchronizing event to occur and then both proceed past the synchronization point. In the non-blocking case, only the receiving process waits while the sender can proceed.

Other approaches to the problem of finding bounds on the separation in time of two events have either been inexact or based on a more restrictive graph topology. Loose bounds that may not enable all possible optimizations were obtained by [Myers & Meng 92] who used the analysis to optimize the implementation of speed-dependent asynchronous circuits (see Section 7.3). Both [McMillan & Dill 92] and [Vanbekbergen et al. 92] handle only acyclic graphs. However, they provide a theoretical foundation upon which our solution is built. Both [Cohen et al. 89] and [Burns 91] use cyclic graph models but they deal only with fixed delays between events.

7.1 Problem Formalization

We could express our concurrent system as an operation-event graph using two different operations for modeling synchronization and computation, e.g., see Figure 7.2. The

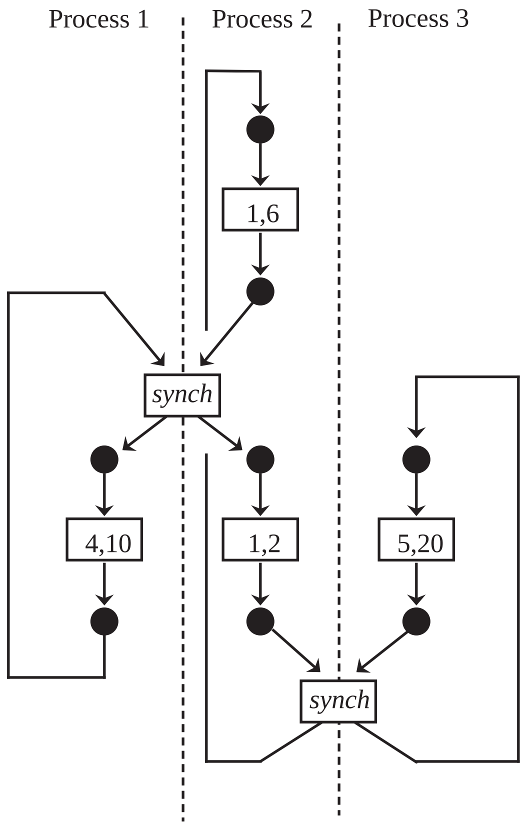


Figure 7.2: The concurrent system of Figure 7.1 expressed as an operation-event graph. The four delay operations have been annotated with lower and upper bounds on the computation time.

separation time analysis could be cast as a verification problem; we need to verify that in any execution of the system the separation in time between related occurrences of two events will not be longer or shorter than a specified value. As discussed in the previous chapter, exact timing verification cannot be accomplished using the methodology described in Section 6.2 because the synchronizations introduce mutual dependencies. Thus the need for the work presented in this chapter, where we develop a body of theory and algorithms that are capable of addressing this fundamental analysis problem. However, due to the inherent simplicity of the model, we will abandon our operation-event graph specification paradigm and use a simpler event-graph representation.

7.1.1 The Process Graph

We represent the system as a directed graph, called the *process graph*, where the vertices represent events (synchronizations) and the edges are annotated with delay information. The process graph for our example (see Figure 7.1) is shown in Figure 7.3.

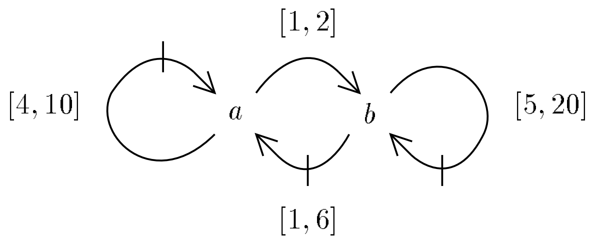


Figure 7.3: The concurrent system of Figure 7.1 expressed as a process graph. The number of lines drawn through an edge indicates the value of the occurrence index offset.

To formalize the problem we use a simple modification of the *event–rule system* developed in [Burns 91][1]. Let $G' = \langle E', R' \rangle$ denote a process graph composed of

- a finite set of (repeatable) events E', the vertices of the graph.
- a finite set of rule templates R', the edges of the graph.

Each edge is labelled with two objects, the delay range $[d, D]$ (integers with $0 \le d \le D$), and the occurrence index offset ε. For our example, we have

$$\begin{aligned} E' &= \{a, b\} \\ R' &= \left\{ a \overset{[4,10],1}{\longmapsto} a,\ a \overset{[1,2],0}{\longmapsto} b,\ b \overset{[1,6],1}{\longmapsto} a,\ b \overset{[5,20],1}{\longmapsto} b \right\}. \end{aligned}$$

[1][Myers & Meng 92] introduced a similarly modified system. The model can also be viewed as an extension of [McMillan & Dill 92] and [Vanbekbergen et al. 92], where we consider cyclic max-only or type-2 graphs. To remain consistent with the notation of [Burns 91] we introduce G' before G, E' before E, etc., i.e., primed variables are introduced before unprimed variables.

The occurrence index offset is used to specify how much the occurrence index is incremented when the edge is executed—see Section 7.1.2. The offset also defines the initial state of the processes, and can be viewed as a marking of the edges that can initially execute. Figure 7.4 shows three identical process graphs with different markings. The rightmost process graph is deadlocked, because the initial marking does not allow any further activity to occur. Note that there is a cycle in this graph without any marked edges.

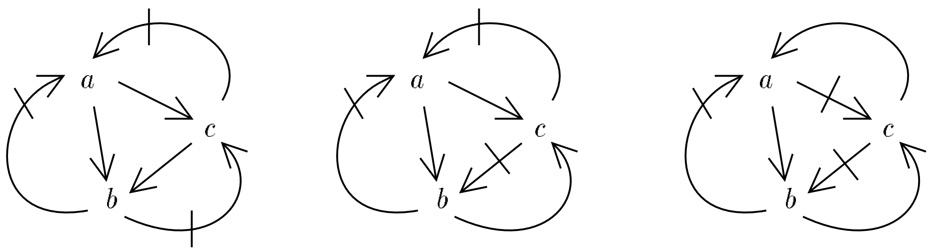


Figure 7.4: Three process graphs with different occurrence index offsets. The offset can be viewed as a marking of the edges that can initially execute. Thus, the leftmost graph would exhibit behavior that can be described by the regular expression $(acb)^*$ because a will occur (both of its incident edges are marked), followed by c, etc. The middle graph exhibits behavior $(abc)^*$ and the rightmost graph is deadlocked.

In the remainder of this chapter, we restrict our analysis to *well-formed* graphs, that is, process graphs that are strongly-connected and have $\varepsilon(c) > 0$ for all cycles c in the graph, where $\varepsilon(c)$ is the sum of the ε values for all edges in the cycle c.

7.1.2 Execution Model

We denote the k^{th} occurrence of event $v \in E'$ as v_k, and refer to k as the *occurrence index* of v_k. Let E be the set of all event occurrences (infinite in one direction, i.e., $k \geq 0$). To model the initial startup behavior of a process, we also include in E a single event occurrence named *root*. Thus,

$$E = \left\{ v_k \,\middle|\, v \in E',\ k \geq 0 \right\} \cup \{root\}.$$

The set R consists of the rules generated by instantiating each rule template of R' at each occurrence index,

$$R = \left\{ u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \;\middle|\; u \overset{[d,D],\varepsilon}{\longmapsto} v \in R',\ k \geq \varepsilon \right\} \cup R_0.$$

Special startup rules are included in the non-empty finite set R_0,

$$R_0 \subset \left\{ root \overset{[d,D]}{\longmapsto} v_k \;\middle|\; v_k \in E - \{root\} \right\}.$$

We call the infinite directed graph constructed from the vertex set E and the edge set R the *unfolded process graph.* Figure 7.5 shows the unfolded process graph for the example in Figure 7.3.

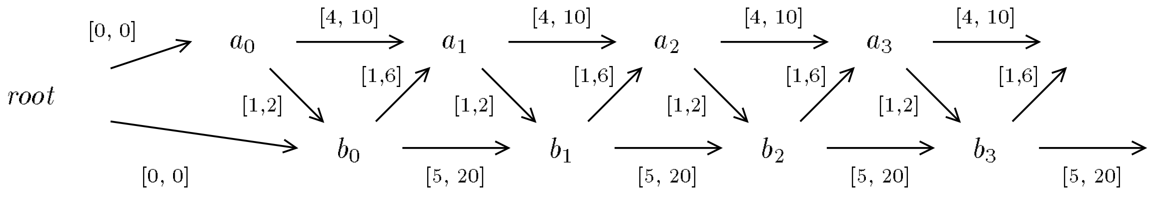


Figure 7.5: A portion of the unfolded process graph for the process graph in Figure 7.3. Two startup edges have been added, specifying that both a_0 and b_0 must occur after time 0.

An *execution* of a process graph is the consistent assignment of time values to event occurrences. A *timing assignment*, τ, maps event occurrences to global time, thus $\tau(v_k)$ is the time of the k^{th} occurrence of event v. The delay information in R restricts the set of possible timing assignments. Formally, we define constraints on the time values introduced by each event occurrence:

$$\begin{aligned} \tau(v_k) &\geq \max\left\{ \tau(u_{k-\varepsilon}) + d \;\middle|\; u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \in R \right\} \\ \tau(v_k) &\leq \max\left\{ \tau(u_{k-\varepsilon}) + D \;\middle|\; u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \in R \right\} \end{aligned}$$

The constraints on $\tau(v_k)$ embody the underlying semantics of a process graph's execution, i.e., events correspond to synchronizations and an event can occur only when all of its incident events have occurred. Each incident event is delayed by some number in a bounded interval ($[d, D]$). Thus, the earliest time at which v_k can occur is constrained by d values, the latest by D values.

7.1.3 Problem Definition

Given two arbitrary events s and t in E' and an integer β, the problem we address is determining the strongest bounds δ and Δ such that for all $\alpha \geq \max(0, \beta)$

$$\delta \leq \ \tau(t_\alpha) - \tau(s_{\alpha-\beta}) \ \leq \Delta.$$

For example, to determine tight bounds on the time separation between two consecutive a events, we would set $s = t = a$ and $\beta = 1$, and consider the bounds on $\tau(a_\alpha) - \tau(a_{\alpha-1})$ for all $\alpha \geq 1$. The bounds are *tight* (i.e., δ is maximal and Δ is minimal), in that they represent separations which could occur during an execution of the system.

We address only the problem of finding the maximum separation Δ, since δ can be obtained from the transformation $\tau(s_\alpha) - \tau(t_{\alpha-(-\beta)}) \leq -\delta$ (i.e., by swapping s and t, negating β, and negating the solution Δ).

7.1.4 Algorithm for a Finite Unfolded Process Graph

We build our solution to this problem on a variation of a graph algorithm developed in [McMillan & Dill 92] that applies only to finite unfolded graphs. In Section 7.2 we will generalize this algorithm to infinite unfolded graphs.

Let Δ_α be the strongest bound for the separation problem given an α, i.e.,

$$\tau(t_\alpha) - \tau(s_{\alpha-\beta}) \leq \Delta_\alpha.$$

We can determine Δ_α by analyzing a finite acyclic graph created by only including the vertices in our unfolded process graph for which there is a path to either t_α or $s_{\alpha-\beta}$. Name the resulting graph $\langle E^*, R^*\rangle$.

The algorithm consists of two simple steps. First, we compute m-values backwards from $s_{\alpha-\beta}$ for all event occurrences,

$$m(v_k) = \max\left\{d(h) \;\middle|\; \text{all paths } v_k \overset{h}{\rightsquigarrow} s_{\alpha-\beta}\right\}$$

where $d(h)$ is sum of the d values of the edges on the path h from v_k to $s_{\alpha-\beta}$ (denoted by $v_k \overset{h}{\rightsquigarrow} s_{\alpha-\beta}$). We can compute these values in linear time in the size of R^* by a reverse topological traversal from $s_{\alpha-\beta}$. If there is no path from v_k to $s_{\alpha-\beta}$, denoted by $v_k \not\rightsquigarrow s_{\alpha-\beta}$, we can assign an arbitrary constant value to $m(v_k)$—we use $m(v_k) = 0$.

We then compute $\Delta_\alpha = M(t_\alpha) - m(t_\alpha)$ by assigning $M(\mathit{root}) = 0$ and then for all other occurrences in (normal) topological order:

$$\begin{array}{l} \text{If } v_k \rightsquigarrow s_{\alpha-\beta} \\ \quad M(v_k) \;=\; \max\left\{ \min\left(M(u_{k-\varepsilon}) + D - m(u_{k-\varepsilon}) + m(v_k)\ ,\ 0\right) \;\middle|\; u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \in R^*\right\} \\ \text{If } v_k \not\rightsquigarrow s_{\alpha-\beta} \\ \quad M(v_k) \;=\; \max\left\{ M(u_{k-\varepsilon}) + D - m(u_{k-\varepsilon}) + m(v_k) \;\middle|\; u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k \in R^*\right\} \end{array} \tag{7.1}$$

Figure 7.6 contains a sample application of the algorithm for the computation of Δ_2 for the example in Figure 7.3. We will not provide a formal proof of this algorithm (see [McMillan & Dill 92]) but do provide an informal explanation.

7.1.5 Informal Justification of the Algorithm

Informally, to maximize the value of $\tau(t_\alpha) - \tau(s_{\alpha-\beta})$ we need to “find an execution” that maximizes $\tau(t_\alpha)$ and minimizes $\tau(s_{\alpha-\beta})$. Consider a special case, namely when $t_\alpha = \mathit{root}$ (see Figure 7.7). By definition we would have $\Delta_\alpha = -m(\mathit{root})$, i.e.,

$$\Delta_\alpha = M(t_\alpha) - m(t_\alpha) = M(\mathit{root}) - m(\mathit{root}) = 0 - m(\mathit{root})$$

In this case, the maximum separation is always non-positive because $m(v_k)$ is always non-negative. This is to be expected, because whenever there is a path from t_α to $s_{\alpha-\beta}$

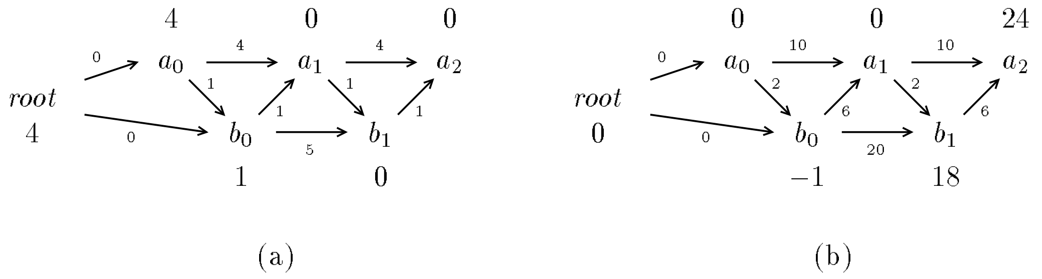


Figure 7.6: Sample execution of the finite acyclic graph algorithm for obtaining Δ_2 for the process graph in Figure 7.3 given the parameters $t = s = a$ and $\beta = 1$, i.e., $s_{\alpha-\beta} = a_1$ and $t_\alpha = a_2$. In (a), the edges are labeled with the d values, and the vertices are labelled with the m-values obtained in the first step of the algorithm. In (b), the edges are labeled with the D values, and the vertices are labelled with the M-values obtained in the second step. We obtain $\Delta_2 = M(a_2) - m(a_2) = 24 - 0 = 24$.

(i.e., $t_\alpha \rightsquigarrow s_{\alpha-\beta}$) then Δ_α will be negative due to the fact that t_α has to occur before $s_{\alpha-\beta}$ can occur. Whenever t_α actually constrains $s_{\alpha-\beta}$ we can compute a maximum separation by fixing $\tau(t_\alpha)$ and minimizing $\tau(s_{\alpha-\beta})$. In this case, since $t_\alpha = root$, we minimize $\tau(s_{\alpha-\beta})$ by picking an execution in which all of the delays are d. If we assume that $\tau(root)$ is zero, then $\tau(s_{\alpha-\beta})$ will be $m(root)$, yielding our result.

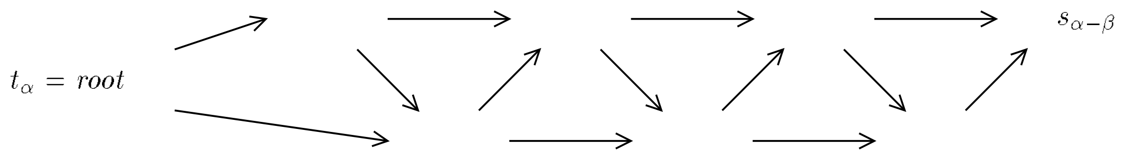


Figure 7.7: To determine Δ_α we essentially maximize $\tau(t_\alpha)$ and minimize $\tau(s_{\alpha-\beta})$. If $t_\alpha = root$, then to minimize $\tau(s_{\alpha-\beta})$ we pick d for all edges and then compute the weight of the longest d path from *root* to $s_{\alpha-\beta}$, i.e., $\Delta_\alpha = -m(root)$. The result is negative because $s_{\alpha-\beta}$ always occurs after *root*.

Now consider another special case, namely when $t_\alpha = v_k$ and v_k is an arbitrary event occurrence except that it has only one incident edge, $u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k$ (see Figure 7.8). In this case, $\Delta_\alpha = M(v_k) - m(v_k)$ and if we assume that $v_k \not\rightsquigarrow s_{\alpha-\beta}$ then (because there is only one incident edge) from Equation 7.1 we have:

$$M(v_k) = M(u_{k-\varepsilon}) + D - m(u_{k-\varepsilon}) + m(v_k)$$

By subtracting $m(v_k)$ from both sides of the equation, we see that this equation is simply stating that:

$$M(v_k) - m(v_k) = M(u_{k-\varepsilon}) - m(u_{k-\varepsilon}) + D \qquad (7.2)$$

In other words, we calculate a maximum separation between $s_{\alpha-\beta}$ and v_k, i.e., $M(v_k) - m(v_k)$, using the maximum separation between $s_{\alpha-\beta}$ and $u_{k-\varepsilon}$, i.e., $M(u_{k-\varepsilon}) - m(u_{k-\varepsilon})$. The separation is changed by D to account for the fact that v_k occurs after $u_{k-\varepsilon}$ and we always want to maximize $\tau(t_\alpha)$.

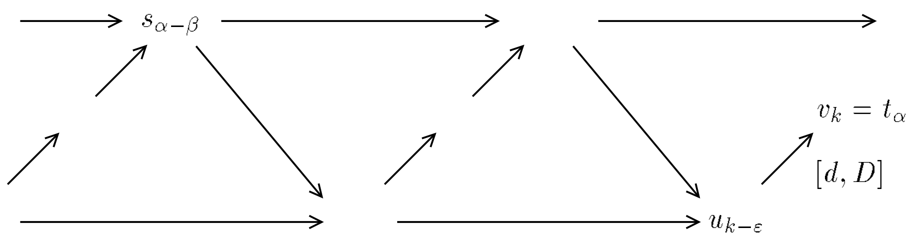


Figure 7.8: To compute Δ_α in this case, we use the maximum separation computed between $s_{\alpha-\beta}$ and $u_{k-\varepsilon}$. Since we always maximize $\tau(t_\alpha)$ we add D (i.e., to compute the maximum separation between $s_{\alpha-\beta}$ and v_k we take into account the maximum delay between $u_{k-\varepsilon}$ and v_k.

If $v_k \rightsquigarrow s_{\alpha-\beta}$ (see Figure 7.9), then it may be that with respect to $u_{k-\varepsilon}$ we cannot delay v_k by D without also delaying $s_{\alpha-\beta}$. If this happens, the maximum separation between $s_{\alpha-\beta}$ and v_k will be $-m(v_k)$. We argued this result when $t_\alpha = root$, i.e., because t_α constrains $s_{\alpha-\beta}$ we simply fix $\tau(t_\alpha)$ and minimize $\tau(s_{\alpha-\beta})$. If v_k does not constrain $s_{\alpha-\beta}$, then the maximum separation between $s_{\alpha-\beta}$ and v_k can be computed from the maximum separation between $s_{\alpha-\beta}$ and $u_{k-\varepsilon}$, i.e., from Equation 7.2. This is precisely what the minimization with zero in Equation 7.1 is accomplishing if $v_k \rightsquigarrow s_{\alpha-\beta}$. When $M(u_{k-\varepsilon}) + D - m(u_{k-\varepsilon}) + m(v_k)$ is positive, v_k constrains $s_{\alpha-\beta}$, and the maximum separation is set to $-m(v_k)$ by setting $M(v_k)$ to zero.

In the general case, we compute $M(v_k)$ by examining the maximum separation that

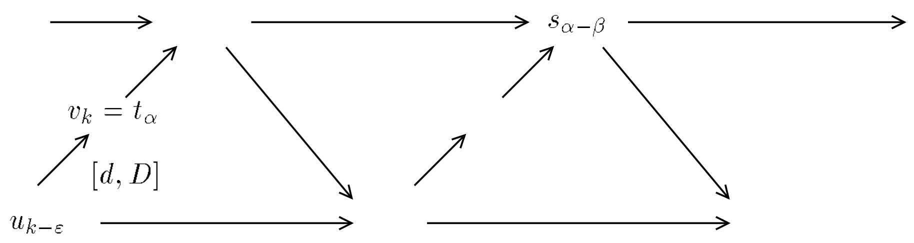


Figure 7.9: To compute Δ_α in this case, we again use the maximum separation computed between $s_{\alpha-\beta}$ and $u_{k-\varepsilon}$ to compute the maximum separation between $s_{\alpha-\beta}$ and v_k (i.e., we add D). However, in this case, we must ensure that the assumption that the delay between $u_{k-\varepsilon}$ and v_k is D is not inconsistent with the calculation of the maximum separation between $s_{\alpha-\beta}$ and $u_{k-\varepsilon}$ (which might assume that it is d).

can be obtained for each occurrence $u_{k-\varepsilon}$ that is incident to v_k. This is the reason for the maximization in Equation 7.1.

7.1.6 Determining Δ

To compute Δ, the maximum separation in time over all event occurrences of s and t, separated in occurrence by β, we maximize Δ_α over all values of α:

$$\Delta = \max\left\{\Delta_\alpha \;\middle|\; \alpha \geq \max(0, \beta)\right\}.$$

Applying the algorithm to the example in Figure 7.3 yields the following maximum separations:

$\tau(a_\alpha) - \tau(a_{\alpha-1}) \leq \Delta_\alpha$

Δ_1	Δ_2	Δ_3	$\Delta_{>3}$
10	24	25	25

and thus $\Delta_\alpha = 25$. The problem, of course, is that this requires an infinite number of applications of the algorithm. Before we present an algebraic solution that allows us to analyze the infinite unfolded graph, we illustrate the difficulties of this analysis with a few examples.

7.1.7 Examples

Our first example, in Figure 7.10, is a process graph that represents two coupled pipelines. If the pipelines were not coupled at c, the maximum separation between a and e would be unbounded. This is because the first pipeline (choosing the delay between consecutive a's as being 2) could be arbitrarily slower than the second pipeline (choosing the delay between consecutive e's as being 1). The coupling of the pipelines forces one pipeline to wait for the other if it gets too far ahead.

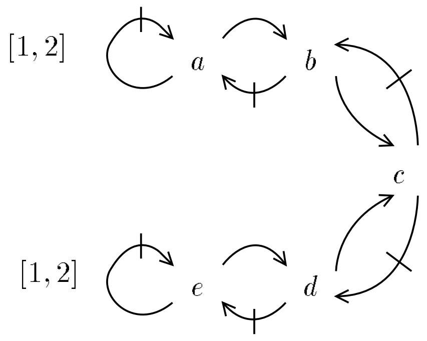


Figure 7.10: A process graph that represents two coupled pipelines. All unspecified delay ranges are $[0, 0]$.

We start the pipeline by *rooting* all of the initial occurrences at zero, i.e., we have startup rules:

$$root \overset{[0,0]}{\longmapsto} a_0, \quad root \overset{[0,0]}{\longmapsto} b_0, \quad root \overset{[0,0]}{\longmapsto} d_0, \quad root \overset{[0,0]}{\longmapsto} e_0$$

For all $\alpha \geq 0$, it can be shown that $\tau(a_\alpha) - \tau(e_\alpha) \leq 4$:

Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	$\Delta_{>4}$
0	1	2	3	4	4

This arises because we can have $\tau(a_0) = 0$, $\tau(a_1) = 2$, $\tau(a_2) = 4$, $\tau(a_3) = 6$, $\tau(a_4) = 8$, $\tau(a_5) = 10$ along with $\tau(e_0) = 0$, $\tau(e_1) = 1$, $\tau(e_2) = 2$, $\tau(e_3) = 3$, $\tau(e_4) = 4$ but we cannot have $\tau(e_5) = 5$ because of the dependency requiring $\tau(e_5)$ to occur no earlier than $\tau(a_3) = 6$. Adding more stages to both pipelines (before the synchronization) would allow e to get even further ahead of a.

Our second example, in Figure 7.11, exhibits interesting behavior. We root all of the initial occurrences at zero. If $\lambda = 6$ then $\tau(a_\alpha) - \tau(a_{\alpha-1}) \leq 8$:

Δ_1	Δ_2	Δ_3	Δ_4	Δ_{odd}	Δ_{even}
4	8	4	8	4	8

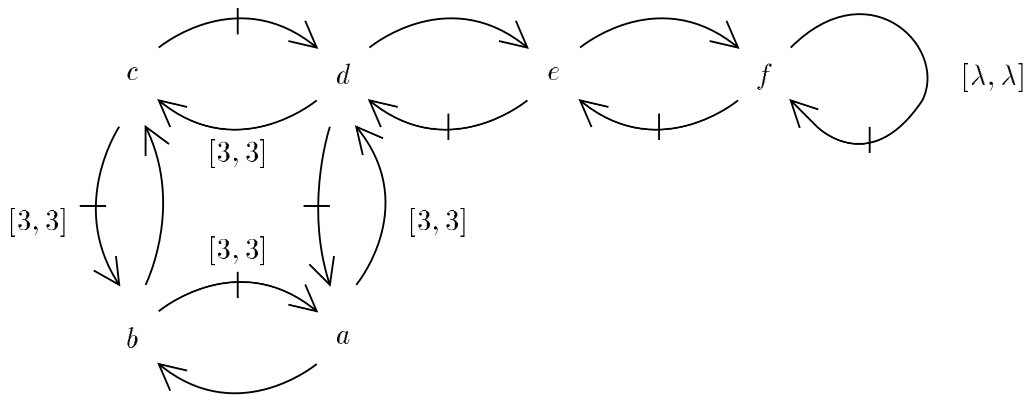


Figure 7.11: A process graph with unusual timing behavior. All unspecified delay ranges are $[1,1]$.

If we change $\lambda = 9$ then $\tau(a_\alpha) - \tau(a_{\alpha-1}) \leq 9$:

Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	$\Delta_{>8}$
4	8	4	8	4	8	8	9	9

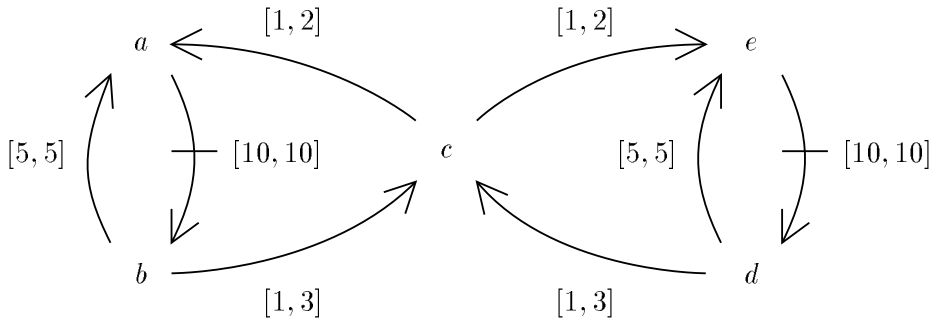


Figure 7.12: Two processes synchronizing at c.

Our final example, in Figure 7.12, corresponds to two simple processes that synchronize at the event c. Clearly, the startup rules can affect the initial timing behavior of the processes. However, this example demonstrates that the initial startup rules also can determine the maximum separation at every point in the infinite execution. We have two startup rules: $root \overset{[\lambda_1,\lambda_1]}{\longmapsto} b_0$ and $root \overset{[\lambda_2,\lambda_2]}{\longmapsto} d_0$ and they determine every Δ_α for

$\tau(e_\alpha) - \tau(a_\alpha)$:

$\Delta_{\geq 0}$			
$\lambda_2 - \lambda_1 \geq 3$	$\lambda_2 - \lambda_1 = 2$	$\lambda_2 - \lambda_1 = 1$	$\lambda_2 \leq \lambda_1$
3	2	1	0

As the process graph is a repetitive system, presumably the Δ_α values will eventually reach a steady state, for example, $\Delta_{\alpha+1} = \Delta_\alpha$ for large α. Unfortunately, as our examples illustrate, the behavior of the Δ_α values can be non-monotonic and periodic, and might even start out periodic and then later stabilize to a constant value. Thus, no simple criteria for determining when steady state has been reached can be derived based on the behavior of the Δ_α values.

7.2 Functional Solution

Our solution to the problem is based on a structural decomposition of the unfolded process graph that exploits its repetitive nature. By dividing the unfolded process graph up into segments and representing the computation of the finite graph algorithm in a symbolic manner we can reuse the computations for each segment.

7.2.1 Introducing Functions

We introduce a symbolic execution of the acyclic algorithm presented in Section 7.1.4. Instead of computing the numeric M-values in Equation 7.1, we compute functions that relate M-values with one another. We present an algebra for representing and manipulating these functions.

Functions are represented as sets of pairs. A singleton set, $\{\langle l, w\rangle\}$, represents the function $f(x) = \min(x + l, w)$. In general, the set

$$\{\langle l_1, w_1\rangle, \langle l_2, w_2\rangle, \ldots, \langle l_n, w_n\rangle\} \tag{7.3}$$

corresponds to the function

$$f(x) = \max\{\min(x + l_i, w_i) \mid 1 \leq i \leq n\}. \tag{7.4}$$

We associate two operators with functions: function maximization, $f \ \underline{max}\ g$, and function composition, $f \circ g$. It follows from Equation 7.4 that function maximization is defined as set union: $f \ \underline{max}\ g = f \cup g$. The following observation leads to an important efficiency optimization:

Pruning Rule Given a function represented as $\{\langle l_1, w_1\rangle, \langle l_2, w_2\rangle, \ldots, \langle l_n, w_n\rangle\}$, if $l_i \geq l_j$ and $w_i \geq w_j$, we can prune the pair $\langle l_j, w_j\rangle$ since for all x, $\min(x + l_i, w_i) \geq \min(x + l_j, w_j)$.

Thus, a function (7.3) can always be represented such that

$$l_1 < l_2 < \ldots < l_n \text{ and } w_1 > w_2 > \ldots > w_n. \tag{7.5}$$

Function composition, $f = g \circ h$, is defined as $f(x) = h(g(x))$. Notice that we use left-to-right function composition [Herstein 64]. For $g = \{\langle l_1, w_1\rangle\}$ and $h = \{\langle l_2, w_2\rangle\}$ we have

$$\begin{aligned}
(g \circ h)(x) &= h(g(x)) = \min(g(x) + l_2, w_2) \\
&= \min(\min(x + l_1, w_1) + l_2, w_2) \\
&= \min(x + l_1 + l_2, \min(w_1 + l_2, w_2)) \\
&= \{\langle l_1 + l_2, \min(w_1 + l_2, w_2)\rangle\}
\end{aligned}$$

If g or h contain more than one pair then

$$\begin{aligned}
g &= g_1 \ \underline{max}\ \cdots \ \underline{max}\ g_n \text{ and} \\
h &= h_1 \ \underline{max}\ \cdots \ \underline{max}\ h_m,
\end{aligned}$$

where g_i and h_i are singleton sets. Function composition is performed using distributivity, i.e.,

$$g \circ h \quad = \quad \underline{max}\ \{g_i \circ h_j \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}\,.$$

We can now express the M-values using functions. We associate a function, f, to each edge $u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k$ in the unfolded process graph: $u_{k-\varepsilon} \overset{f}{\longmapsto} v_k$, where

$$f \;=\; \begin{cases} \{\langle D - m(u_{k-\varepsilon}) + m(v_k), 0\rangle\} & \text{if } v_k \rightsquigarrow s_{\alpha-\beta} \\ \{\langle D - m(u_{k-\varepsilon}) + m(v_k), \infty\rangle\} & \text{if } v_k \not\rightsquigarrow s_{\alpha-\beta} \end{cases}$$

The function f incorporates the min-part of Equation 7.1, and the max-part of Equation 7.1 corresponds to function maximization of the functions for the incoming edges. Using function composition and function maximization, we can create a function F_{v_k} that relates $M(\mathit{root})$ to $M(v_k)$, i.e., $M(v_k) = F_{v_k}(M(\mathit{root}))$, where

$$F_{v_k} \;=\; \underline{max} \;\{\; F_{u_{k-\varepsilon}} \circ f \mid u_{k-\varepsilon} \overset{f}{\longmapsto} v_k \in R^* \;\}$$

$$\begin{tikzcd}
 & a_0 \arrow[rr, "f_4"] \arrow[ddr, "f_3"] & & a_1 \arrow[rr, "f_7"] \arrow[ddr, "f_8"] & & a_2 \\
\mathit{root} \arrow[ur, "f_1"] \arrow[drr, "f_2"'] & & & & & \\
 & & b_0 \arrow[uur, "f_5"'] \arrow[rr, "f_6"'] & & b_1 \arrow[uur, "f_9"'] &
\end{tikzcd}$$

Figure 7.13: Fragment of unfolded process graph annotated with functions corresponding to each edge (the m-values are given in Figure 7.6 (a)).

For the example in Figure 7.6 (see Figure 7.13), we relate $M(\mathit{root})$ to $M(b_0)$ with the function

$$\begin{aligned} F_{b_0} &= f_1 \circ f_3 \;\underline{max}\; f_2 \\ &= \{\langle 0,0\rangle\} \circ \{\langle -1,0\rangle\} \;\underline{max}\; \{\langle -3,0\rangle\} \\ &= \{\langle -1,-1\rangle, \langle -3,0\rangle\} \end{aligned}$$

Evaluating the function (using Equation 7.4) at $M(\mathit{root}) = 0$ yields

$$F_{b_0}(0) = \max(\min(0 + -1, -1), \min(0 + -3, 0)) = -1$$

which is exactly the value obtained for $M(b_0)$ in Figure 7.6 (b). The functions F_{b_0} and F_{a_0} are then used to relate $M(\mathit{root})$ to $M(a_1)$, etc., until a function that relates

$M(root)$ to $M(t_\alpha)$ is created. In our example, $t_\alpha = a_2$ and the construction produces $F_{a_2} = \{\langle 22, 25\rangle, \langle 24, 24\rangle\}$.

We can find the separation between $s_{\alpha-\beta}$ and t_α as $\Delta_\alpha = M(t_\alpha) - m(t_\alpha)$ where $M(t_\alpha) = F_{t_\alpha}(M(root)) = F_{t_\alpha}(0)$. For our example, we get $\Delta_2 = F_{a_2}(0) - 0 = 24$, where F_{a_2} is evaluated using Equation 7.4.

7.2.2 Decomposition

Instead of forming a single function relating $M(root)$ to $M(t_\alpha)$, we can perform this construction in segments, that is, determine the functional relationship between $M(root)$ and the M-values at some interior nodes, and compose those functions with the functions relating the M-values at the interior nodes with $M(t_\alpha)$. We will see that this process is akin to matrix multiplication.

Consider an unfolded process graph used to determine Δ_α. We decompose the graph into three segments: an initial segment, **R**, containing the *root* event, a terminal segment, **T**, containing $s_{\alpha-\beta}$ and t_α, and an interior segment, **S** (see Figure 7.14 (a)).

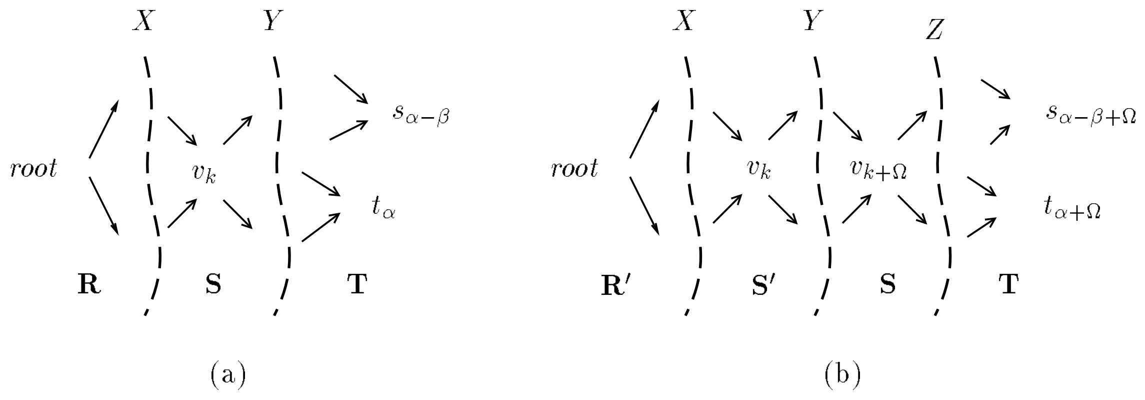


Figure 7.14: Decomposing an unfolded process graph into segments.

A *cutset* is a set of event occurrences such that every path from the *root* to t_α goes through an element of the cutset. Let X and Y be two cutsets such that

$$v_k \in X \text{ if and only if } v_{k+\Omega} \in Y.$$

We say that Y is X *shifted to the right* by Ω. We can construct a square matrix $\mathbf{S}$ that maps the M-values of the events in X to the M-values of the events in Y, i.e., to the same events Ω occurrences later ($\Omega > 0$). Similarly, we can construct a matrix $\mathbf{R}$ that maps $M(root)$ to the M-values of the events in X, and a matrix $\mathbf{T}$ that maps the M-values of the events in Y to $M(t_\alpha)$. We can now restate the maximum separation problem in matrix form. Using ($\underline{max}$, $\circ$) matrix multiplication, that is, function maximization for scalar addition, and function composition for scalar multiplication, we can form $\mathbf{RST}$, a 1×1 matrix containing a single function relating $M(root)$ to $M(t_\alpha)$, which is used to obtain Δ_α.

For the graph in Figure 7.13, a possible decomposition is $X = \{a_0, b_0\}$ and $Y = \{a_1, b_1\}$ yielding

$$\mathbf{RST} = \begin{pmatrix} f_1 & f_1 \circ f_3 \ \underline{max}\ f_2 \end{pmatrix} \begin{pmatrix} f_4 & f_4 \circ f_8 \\ f_5 & f_5 \circ f_8 \ \underline{max}\ f_6 \end{pmatrix} \begin{pmatrix} f_7 \\ f_9 \end{pmatrix}$$

Now consider finding $\Delta_{\alpha+\Omega}$. We add another $\mathbf{S}$ segment to the graph, defined by the cutsets Y and Z, where Z is Y shifted to the right by Ω (see Figure 7.14 (b)). We get the matrix product $\mathbf{R}'\mathbf{S}'\mathbf{ST}$ where $\mathbf{S}$ and $\mathbf{T}$ are the same as above, but $\mathbf{R}'$ and $\mathbf{S}'$ may differ from $\mathbf{R}$ and $\mathbf{S}$ since the m-values are now computed from $s_{\alpha-\beta+\Omega}$ instead of $s_{\alpha-\beta}$. This decomposition is only useful if we can arrange the symbolic computation such that $\mathbf{R}' = \mathbf{R}$ and $\mathbf{S}' = \mathbf{S}$, i.e., such that adding an $\mathbf{S}$ segment *will not* change the functional representation. The next section characterizes the behavior of the m-values that allows us to utilize this decomposition effectively.

7.2.3 Repetition of the m-values

Since the m-values are constructed from a repetitive system (the process graph) the values eventually are determined by the *maximum ratio cycles* in the process graph. A maximum ratio cycle c is a cycle with ratio $d(c)/\varepsilon(c)$ equal to that of the maximum ratio $r^\star$:

$$r^\star = \max_{c \text{ a simple cycle in } G'} \frac{d(c)}{\varepsilon(c)}.$$

The behavior of the m-values is best illustrated by a classical graph problem [Lawler 76].

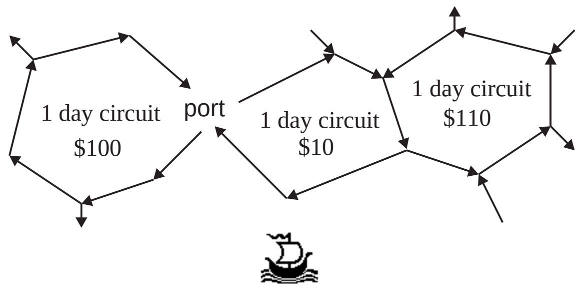


Figure 7.15: Maximize profit over an n-day voyage.

Consider a cargo steamer whose purpose is to maximize profit over an n-day voyage. In the graph of Figure 7.15, the cycle to the right is the one that maximizes the profit per day, however, to get there and return, the steamer needs to spend a day in the low-profit cycle of \$10. Thus for short trips ($n < 10$) it is more profitable to use the leftmost cycle, and for long trips ($n > 10$) it is worth suffering a "low-income" day so that more profit can be obtained using the rightmost cycle. The break-even point is for a 10 day voyage. To relate this to the m-values, note that for a given cycle, c, in the graph, the profit corresponds to $d(c)$, the number of days is $\varepsilon(c)$, and $r^\star$ is the maximum profit obtainable per day. The maximum profit for the entire voyage ($m(v_k)$) is computed assuming that the cargo steamer leaves one port (the event s) and ends the voyage at a new port (the event v), and maximizes profit over n-days (where $n = \alpha - \beta - k$, the number of unfoldings required to reach v_k from $s_{\alpha-\beta}$).

The m-values *repeat* precisely when the values for *all* events are determined repetitively using maximum ratio cycles. Formally, there exists integers $k^\star$ and $\varepsilon^\star$ such that for all k, $\varepsilon^\star \leq k \leq \alpha - k^\star$ and all $v \in E'$

$$m(v_{k-\varepsilon^\star}) - m(v_k) = r^\star \varepsilon^\star, \tag{7.6}$$

where $k^\star$ is the number of unfoldings of the process graph (relative to $s_{\alpha-\beta}$) before all of the m-values repeat and $\varepsilon^\star$ is the occurrence period of this repetition. If there are multiple cycles with maximum ratio then the m-values computed for different events may use different maximum ratio cycles. Thus, a simple upper bound on $\varepsilon^\star$ is the least common multiple of $\varepsilon(c)$ for each maximum ratio cycle c.

Figure 7.16 illustrates the behavior of the m-values for the process graph in Figure 7.3. Both $k^\star$ and $\varepsilon^\star$ are values specific to a particular process graph. For example, changing the delays $[4,10]$ and $[5,20]$ to [999,1000] and [1000,1000], respectively, changes $k^\star$ from 3 to 998. Note that only the lower delay bounds affect $k^\star$.

$$\begin{tikzcd}
& 22 & & 17 & & 12 & & 8 & & 4 & & 0 & & \\
{} \arrow[r] & a_5 \arrow[rr] \arrow[dr] & & a_6 \arrow[rr] \arrow[dr] & & a_7 \arrow[rr] \arrow[dr] & & a_8 \arrow[rr] \arrow[dr] & & a_9 \arrow[rr] \arrow[dr] & & a_{10} \arrow[rr] \arrow[dr] & & {} \\
{} \arrow[rr] & & b_5 \arrow[rr] \arrow[ur] & & b_6 \arrow[rr] \arrow[ur] & & b_7 \arrow[rr] \arrow[ur] & & b_8 \arrow[rr] \arrow[ur] & & b_9 \arrow[rr] \arrow[ur] & & {} & \\
& & 21 & & 16 & & 11 & & 6 & & 1 & & &
\end{tikzcd}$$

Figure 7.16: A portion of the unfolded process graph for the process graph in Figure 7.3 labeled with m-values ($s_{\alpha-\beta} = a_{10}$). The m-values repeat when $m(a_6) - m(a_7) = 5$ which occurs after three unfoldings relative to a_{10}, thus $k^\star = 3$. The occurrence period of the repetition is one, making $\varepsilon^\star = 1$.

7.2.4 Identical S Matrices

After $k^\star$ unfoldings of the process graph, the m-values are repeating. Let $\mathbf{T}$ be the matrix obtained from the cutsets $X_{k^\star}$ and $\{t_\alpha\}$, where cutset $X_{k^\star}$ has the property that the m-values for the vertices topologically left of $X_{k^\star}$ repeat (with an occurrence period of $\varepsilon^\star$). Thus, Equation 7.6 implies that for all edges $u_j \overset{[d,D]}{\longmapsto} v_k$ where v_k is topologically left of $X_{k^\star}$:

$$m(u_{j-\varepsilon^\star}) - m(v_{k-\varepsilon^\star}) = m(u_j) - m(v_k).$$

This makes the M-values independent of k. Therefore, after $k^\star$ unfolding of the process graph (relative to $s_{\alpha-\beta}$), the functional representations of $\mathbf{R}$ and $\mathbf{S}$ remain the same independently of the number of unfoldings.

Let the matrix product $\mathbf{RT}$ solve $\Delta_{\alpha^\star}$. We can find $\Delta_{\alpha^\star+\varepsilon^\star}$ by adding an $\mathbf{S}$ segment, i.e., **RST**. By repeatedly adding **S** segments to the graph, we can compute $\Delta_{\alpha^\star + n\varepsilon^\star}$ for $n \geq 0$ from $\mathbf{R}\underbrace{\mathbf{SS}\cdots\mathbf{S}}_{n}\mathbf{T}$. The maximum over all $n \geq 0$ can be found from

$$\mathbf{RT}\ \underline{max}\ \mathbf{RST}\ \underline{max}\ \mathbf{RSST}\ \underline{max}\ \ldots$$

which by matrix algebra can be rewritten as

$$\mathbf{R}(\mathbf{I}\ \underline{max}\ \mathbf{S}\ \underline{max}\ \mathbf{S}^2\ \underline{max}\ \mathbf{S}^3\ \underline{max}\ \ldots)\mathbf{T}, \tag{7.7}$$

where **I** is the identity matrix. The elements of **I**, $\overline{0}$ and $\overline{1}$, are the identity elements for function maximization and composition, respectively. We have $\overline{0} = \{\langle -\infty, -\infty \rangle\}$ and $\overline{1} = \{\langle 0, \infty \rangle\}$ (note that $\overline{0}$ is an annihilator for function composition).

A matrix closure algorithm [Aho et al. 74] can be used to compute $\mathbf{S}^*$, the middle part of (7.7), because in this context, function maximization and composition form a *closed semi-ring.* This is the key observation that allows us to implicitly compute an infinite number of Δ_α values.

To compute $\mathbf{S}^*$ we need to be able to compute the closure of the diagonal elements of **S**. For $f = \{\langle l_1, w_1 \rangle, \ldots, \langle l_n, w_n \rangle\}$, the scalar closure operation

$$f^* = \overline{1}\ \underline{max}\ f\ \underline{max}\ f^2\ \underline{max}\ f^3\ \underline{max}\ \ldots$$

can be efficiently computed by:

$$f^* = \begin{cases} \{\overline{1}, \langle \infty, w_q \rangle\} & \text{if } l_n > 0 \\ \{\overline{1}\} & \text{if } l_n \leq 0 \end{cases}$$

where the pairs are ordered as in (7.5) and w_q corresponds to the first positive l, i.e., $l_q > 0$ and if $q > 1$ then $l_{q-1} \leq 0$. We can form the closure of an $n \times n$ matrix in $O(n^3)$ scalar semi-ring operations ($n = O(E')$).

$\mathbf{RS}^*\mathbf{T}$ is used to compute the maximum of the Δ_α values for only a subset of the integers $\alpha \geq \max(0, \beta)$. If $\varepsilon^\star = 1$, we need only compute the maximum of a finite number of additional Δ_α, precisely for those $\alpha < \alpha^\star$, since **RT** is used to compute $\Delta_{\alpha^\star}$. This is done by applying the finite graph algorithm for each α such that $\max(0, \beta) \leq \alpha < \alpha^\star$.

If $\varepsilon^\star > 1$, we need to also compute those α such that $\varepsilon^\star$ does not divide $\alpha - \alpha^\star$. This can be accomplished by choosing $\varepsilon^\star$ different initial matrices, named $\mathbf{R}_0$, $\mathbf{R}_1$, …, $\mathbf{R}_{\varepsilon^\star-1}$, corresponding to $0, 1, \ldots, \varepsilon^\star - 1$ additional unfoldings of the process graph. Thus we can compute the maximum of Δ_α for all $\alpha \geq \alpha^\star$ by creating the function

$$(\mathbf{R}_0 \ \underline{max} \ \mathbf{R}_1 \ \underline{max} \ \ldots \ \underline{max} \ \mathbf{R}_{\varepsilon^\star-1})\mathbf{S}^*\mathbf{T}$$

7.2.5 Example

We now apply the details of the decomposition method to the example in Figure 7.3. We decompose the unfolded process graph into matrices **R**, **S**, and **T** as shown in Figure 7.17. The size of the **T** segment is determined as $k^\star = 3$ unfoldings relative to the $s_{\alpha-\beta}$ node, and the size of the **S** segment is $\varepsilon^\star = 1$ unfoldings. The functions in **S** relate $M(a_0)$ and $M(b_0)$ to $M(a_1)$ and $M(b_1)$. For this example

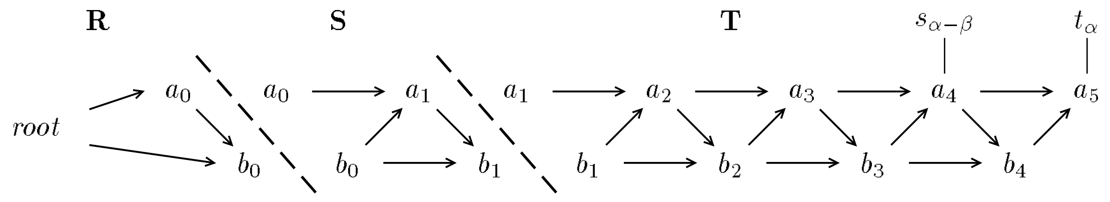


Figure 7.17: A decomposed unfolded process graph corresponding to the process graph in Figure 7.3.

$$\mathbf{R} = \begin{pmatrix} \{\langle 0,0\rangle\} & \{\langle 1,0\rangle\} \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} \{\langle 5,0\rangle\} & \{\langle 6,0\rangle\} \\ \{\langle 2,0\rangle\} & \{\langle 15,0\rangle\} \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} \{\langle 46,25\rangle\} \\ \{\langle 55,25\rangle\} \end{pmatrix}$$

The closure of **S** is:

$$\mathbf{S}^* = \begin{pmatrix} \{\overline{1}, \langle\infty,0\rangle\} & \{\langle\infty,0\rangle\} \\ \{\langle\infty,0\rangle\} & \{\overline{1}, \langle\infty,0\rangle\} \end{pmatrix}$$

yielding the final product

$$\mathbf{RS}^*\mathbf{T} = (\{\langle\infty, 25\rangle\}).$$

The maximum separation between $a_{\alpha-1}$ and a_α for $\alpha \geq 4$ is computed from the function $f = \{\langle\infty, 25\rangle\}$, i.e., $\Delta_{\geq 4} = f(M(root)) - m(a_5) = f(0) - 0$, yielding 25.

7.2.6 Efficiency Considerations

There are two potential inefficiencies associated with this algorithm.

1. Both $\varepsilon^\star$ and $k^\star$ depend on the delay ranges and are not polynomial in the size of the process graph.

2. The size of the representation of a particular function may be as large as the number of paths between the two events related by the function.

Point 1 is potentially serious, however in most process graphs derived from circuits, $\varepsilon^\star = 1$ (see [Burns 91]). $k^\star$ is more of a concern because it can be large if there exists a cycle c such that $d(c)/\varepsilon(c)$ is almost equal to $r^\star$. Although of theoretical interest, point 2 is not likely to be of practical concern. In practice the functions can be efficiently pruned and the size of the functions seems to be linear with respect to the size of the process graph.

7.3 Practical Applications

This section describes two applications demonstrating the practicality of the algorithm for realistic examples.

7.3.1 Memory Management Unit

Consider an edge $u_{k-\varepsilon} \overset{[d,D]}{\longmapsto} v_k$ in an arbitrary process graph. If the minimum time separation between $u_{k-\varepsilon}$ and v_k is *larger* than D, event $u_{k-\varepsilon}$ will never constrain the time of event v_k, i.e., v_k must always wait for some other event to occur, and the edge from $u_{k-\varepsilon}$ can be removed from the process graph without changing the behavior of the system.

This idea can be used to remove redundant circuitry in asynchronous circuits given (conservative) bounds on the actual delays of a speed-independent design. Superfluous edges can be removed by analyzing the process graph corresponding to the circuit. This approach has been taken by Myers and Meng [Myers & Meng 92] who use an inexact timing analysis algorithm, i.e., the algorithm doesn't necessarily give tight bounds on separation times. Clearly, being able to obtain tight bounds potentially enables the removal of more edges.

One of the examples in [Myers & Meng 92] is a memory management unit (MMU) designed to interface to the Caltech Asynchronous Microprocessor [Martin et al. 89]. The process graph (for one of the possible execution modes of the MMU) consists of 16 events and 23 edges and is shown in Figure 7.18.

For the chosen delay intervals, $k^\star = 1$ and $\varepsilon^\star = 1$. Analyzing the 23 edges using our exact algorithm takes on average .1 CPU seconds on a SPARC 2 for each edge. The analysis results in the removal of six edges from the process graph (the dotted edges in Figure 7.18) or equivalently, the removal of six transistors from the circuit. This is the same result as in [Myers & Meng 92].

7.3.2 Asynchronous Microprocessor

A subset of the Caltech Asynchronous Microprocessor [Martin et al. 89] has been modelled and analyzed using the techniques described in this chapter (see [Hulgaard et al. 93]). The process graph for this simplified model consists of 60 events and 127 edges, and has $\varepsilon^\star = 1$ and $k^\star \leq 3$. Computations of the instruction fetch cycle period and the pipeline latency can be performed in under 2 CPU seconds on a SPARC 2.

These and similar computations can be used to determine the real-time properties of the asynchronous microprocessor. For example, to bound the execution time of a code fragment, we can use the minimum and maximum separation in time for each instruction

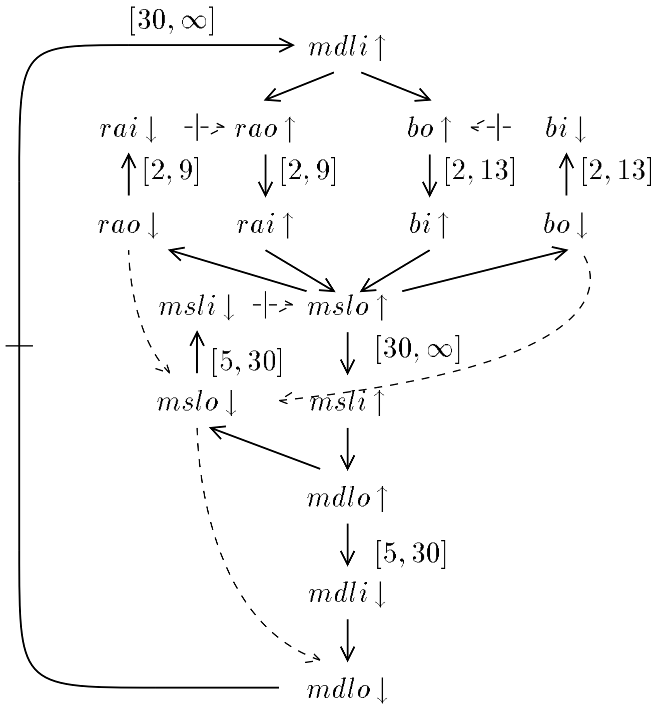


Figure 7.18: Process graph for memory management unit (from [Myers & Meng 92]). All unmarked edges have [0, 1] as the delay range. The dotted edges are superfluous and can be removed without affecting the behavior of the system.

type [Park & Shaw 91]. Furthermore, this information is useful when interfacing the microprocessor to an external synchronous component, especially in cases where the synchronous component is clocked using a signal produced by the microprocessor.

7.4 Discussion

In this chapter, we have presented an efficient and exact solution to a fundamental problem in circuit synthesis and optimization, namely, the determination of bounds on the separation in time of events in concurrent systems without conditional or iterative behavior. The major contribution of this work is the structural decomposition of the infinitely unfolded process graph that enables it to be implicitly analyzed to obtain the tightest possible bounds. This aspect of our algorithm and its algebraic formulation enables it to be efficient enough for practical use. Furthermore, our algorithm handles a wide range of process graphs and is thus useful in a variety of domains. We presented two applications in the area of asynchronous circuit synthesis and analysis. Several other practical applications have been identified, and are discussed in [Hulgaard et al. 93]. Because this work addresses a fundamental problem in the analysis of concurrent systems, i.e., determining how synchronization affects temporal behavior, it may serve as a foundation upon which other analysis tools can be built.

Chapter 8

Conclusions and Contributions

Timing behavior is an important yet somewhat neglected area of research in design automation, especially at high levels of abstraction. Temporal behavior needs to be specified and analyzed if design automation tools are to deal with higher levels of abstraction and exploit guarantees regarding temporal performance. This dissertation has focused precisely on these two problems: providing a formal specification for the timing behavior of systems (primarily hardware) described at many different levels of abstraction, and exploring the temporal analysis of abstract concurrent systems.

This chapter is divided into two sections. The first summarizes the work presented in this dissertation, and the second describes topics for future research.

8.1 Summary of Contributions

In this Section, we present a summary of the contributions of this dissertation with respect to all four areas of design automation: specification, validation, synthesis, and verification.

8.1.1 Specification

The event-based specification paradigm is a natural and appealing model upon which to base a specification language for timing behavior. In summary, the primary contribution of this dissertation in the area of timing specification is the development and presentation of a new model, the *operation-event graph*, a generalization of the event-graph specification paradigm. This work has been motivated by the recognition that timing constraints are relationships between discrete events, and should not be represented simply as edges in an event graph. Key features of the representation include:

- The use of *event-logic*, a natural yet expressive language for constraining temporal behavior. The logic relies on both causal and chronological relationships to identify the discrete events being constrained.

- The incorporation of structure (wires) into an event-based specification language. Designs can thus be specified at many different levels of abstraction, using the same specification language.

- The flexibility provided by the use of an operational semantics for functionality and a denotational semantics for constraining temporal behavior (of course, as was pointed out in Section 3.3, there is no clean distinction between functional and temporal behavior).

8.1.2 Validation

Our specification language was designed with simulation in mind, and a clear simulation semantics was a requirement for all features of the model. The ability to execute a specification is of utmost importance because users need to be able to validate their specifications.

The simulator, OEsim, was created to demonstrate that an event-based specification language could be both very expressive and also support executability of the specification. The key features of the simulator include:

- The ability to report timing constraint violations; i.e., to detect inconsistencies between the operational and denotational specifications. This is accomplished efficiently through the use of incremental constraint checking and a variety of optimizations.
- Support for modularity and encapsulation. Timing constraints can be encapsulated within specifications that can be multiply instantiated into one or more designs. Validation of temporal correctness can then naturally take place when the design is simulated to check its functionality.
- Support for rapid prototyping, especially when temporal relationships must be specified and taken into account. Few other existing behavioral simulators provide such extensive support for timing constraint specification and validation.

8.1.3 Synthesis

Few direct contributions to the area of synthesis have been presented in this dissertation. However, if synthesis tools are to take timing behavior into account, they will need specification languages that are capable of formally expressing complex behavior. Moreover, synthesis algorithms will need to make use of the timing information that can be provided by verification. Traditionally, verification is thought of as a post-synthesis step, in which an implementation is verified with respect to a specification. In this dissertation, verification is viewed quite differently, in that it provides guarantees regarding temporal behavior, and these guarantees are quite useful for synthesis. For example, in [Amon & Borriello 91b], we present a synthesis task, the sizing of synchronization queues for concurrent systems, based entirely upon timing information provided either manually by the user or automatically by verification tools.

In summary, although this dissertation is not directed towards synthesis, many of the contributions will facilitate the development of new design automation tools and algorithms for synthesis. For example, the verification algorithm of Chapter 7 can be used to optimize the synthesis of asynchronous circuits, simplify combinational and sequential logic by extracting temporal don't care information, and focus optimization efforts in data-path synthesis by generating useful scheduling constraints (see [Hulgaard et al. 93]).

8.1.4 Verification

Timing verification is a very difficult problem, and often NP-complete even when both the functionality and the timing constraints are specified using a very simple and restrictive semantics (e.g., see [McMillan & Dill 92]). In this dissertation, we have presented two theoretically exponential algorithms for verifying the timing behavior of concurrent systems. However, in practice, both algorithms are quite efficient, and represent major contributions.

Chapter 6 presented a symbolic timing verifier that allows delays and constraints to be expressed as variables. This powerful extension to traditional constraint checking provides insight into how different aspects of a design relate to one another (with respect to constraint satisfaction) because the results of verification are descriptions of the relationships between variables that need to be satisfied for verification to be successful. Moreover, the techniques can also provide guarantees regarding the temporal behavior of the modeled system. The verification model allows conditional and parallel behavior as well as limited forms of concurrency. The feasibility of performing timing verification using symbolic variables has been demonstrated.

Chapter 7 presented an efficient and exact algorithm for determining exact bounds on the separation in time of two arbitrary system events. This problem is fundamental in that it addresses a core issue, determining how synchronization affects the temporal behavior of concurrent systems. We are not aware of any other existing tools or formalisms capable of performing the types of analysis that can now be performed as a result of this work.

8.2 Directions for Future Research

Much of the work described in this dissertation has been driven by the goal of extending expressivity. In the area of specification, a new event-based specification paradigm was presented. In the area of verification, new methodologies and a new fundamental problem were addressed. As one would expect, future research could undoubtedly attempt to extend even further the expressiveness of the models we have described. For example, none of the models that were presented support true hierarchy (OEgraphs support modularity, but are not fully hierarchical), which should ideally be incorporated into all of our models.

In the area of specification, however, our model appears to be sufficiently expressive. A much more interesting problem is in fact determining how to best restrict the model. Simple event-graphs are clearly not expressive enough, because there are many common constraints appearing in databooks that they cannot handle. These constraints can be specified using operation-event graphs and event-logic, but both may be overly-expressive with respect to most designers needs. The chronological and causal relationships we used to identify the constrained discrete events may not always be necessary. Simpler models may be more appropriate in that they will permit more automation (i.e., in synthesis and verification) and yet still satisfy the needs of designers.

In the area of verification, we would clearly like to be able to verify more complicated systems than those described in Chapter 7. At present, systems expressed as communicating pieces of software cannot effectively be analyzed because we do not handle conditional behavior (e.g., programs have "if" statements). This model needs to be extended to handle conditional and iterative behavior. However, when communication (i.e., synchronization) is itself conditional, the problem appears to be quite complicated. In this case, it may be necessary to relax the tightness of the bounds in favor of computational efficiency. It would also be desirable to integrate the symbolic verification methodology of Chapter 6 with the verification algorithms of Chapter 7. At present, applications for the symbolic timing verifier are limited due to its inability to handle the types of concurrency described in Chapter 7. The verifier represents a proof of concept, and further theoretical work is needed. Some of the restrictions to the model described in Chapter 7 can undoubtedly be removed. For example, it should be possible to analyze graphs that are not strongly connected. Further work is needed in order to establish a lower bound on the complexity of the problem.

Looking further ahead, newer technologies (e.g., field programmable gate arrays) and the growing popularity of programmable micro-controllers have served to blur the traditional boundary between hardware and software. Many new systems will likely be implemented using combinations of both hardware and software, and research in the area of Hardware/Software Co-Design is in its infancy. Synthesis algorithms for these systems must take timing behavior into account—to facilitate optimizations (using the temporal guarantees obtained from timing verification), to provide information to help with partitioning and scheduling, and to preserve correctness.

8.3 Closing Comments

This dissertation has addressed the problem of specifying and verifying correct temporal behavior for digital and possibly other concurrent systems. It has expanded the frontiers of specification, and looked beyond the simple event-graph. The benefits of performing symbolic timing verification have been demonstrated, namely, that synthesis and analysis algorithms can be cast into this framework. Finally, we have examined a fundamental problem in the area of analyzing concurrent systems, and presented a theoretical framework and efficient algorithms which may have a wide domain of applicability.

Bibliography

[Aho et al. 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[Ajmone Marsan 89] M. Ajmone Marsan. Stochastic Petri nets: An elementary introduction. In G. Rozenberg, editor, *Advances in Petri Nets 1989*, number 424 in Lecture Notes in Computer Science, pages 1–29. Springer–Verlag, 1989.

[Allen 83] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), November 1983.

[Alur & Dill 90] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages, and Programming*. Springer-Verlag Lecture Notes in Computer Science 443, 1990.

[Alur & Henzinger 89] R. Alur and T. A. Henzinger. A really temporal logic. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, 1989.

[Alur & Henzinger 90] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[Alur & Henzinger 92] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, K. Huizing, W. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*. Springer-Verlag Lecture Notes in Computer Science 600, 1992.

[Alur et al. 89] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. In *Proceedings of the 10th Principles of Distributed Computing*, 1989.

[Amon & Borriello 91a] T. Amon and G. Borriello. OEsim: A simulator for timing behavior. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[Amon & Borriello 91b] T. Amon and G. Borriello. Sizing synchronization queues: A case study in higher level synthesis. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[Amon & Borriello 92] T. Amon and G. Borriello. An approach to symbolic timing verification. In *29th ACM/IEEE Design Automation Conference.* ACM/IEEE, June 1992.

[Amon et al. 91] T. Amon, G. Borriello, and C. Séquin. Operation/Event graphs: A design representation for timing behavior. In *1991 IFIP Conference on Hardware Description Languages (CHDL)*, 1991.

[Amon et al. 93] T. Amon, H. Hulgaard, S. Burns, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. In *IEEE International Conference on Computer Design (ICCD)*, October 1993.

[Andre 91] C. Andre. Delays in synchronized elementary net systems. In G. Rozenberg, editor, *Advances in Petri Nets 1991*, number 524 in Lecture Notes in Computer Science. Springer–Verlag, 1991.

[Arnold 85] J. Arnold. The knowledge-based test assistant's wave/signal editor: An interface for the management of timing constraints. In *Second Conference on Artificial Intelligence Applications*, December 1985.

[Augustin 89] L. M. Augustin. An algebra of waveforms. In L. Claesen, editor, *Proceedings of the IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, pages 159–168, Leuven, Belgium, November 1989. North-Holland.

[Augustin et al. 88] L. Augustin, B. Gennart, Y. Huh, D. Luckham, and A. Stanculescu. An overview of VAL. Technical Report CSL-TR-88-367, Stanford University, 1988.

[Barbacci 81] M. Barbacci. Instruction set processor specification (ISPS): The notation and its applications. *IEEE Transactions on Computers*, January 1981.

[Bennett 86] M. J. Bennett. *Proving Correcness of Asynchronous Circuits Using Temporal Logic.* PhD dissertation, University of California at Los Angeles, April 1986.

[Bestavros 90] A. A. Bestavros. The input output timed automaton: A model for real-time parallel computation. *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[Bochmann 82] G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3), March 1982.

[Borriello 88a] G. Borriello. Combining event and data-flow graphs in behavioral synthesis. *Proceedings of the International Conference on Computer Aided Design*, November 1988.

[Borriello 88b] G. Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis.* PhD dissertation, University of California at Berkeley, 1988.

[Brauer et al. 87] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Advances in Petri Nets 1986.* Number 254 in Lecture Notes in Computer Science. Springer–Verlag, 1987. Part I: *Petri Nets: Central Models and their Properties*, Part II: *Petri Nets: Applications and Relationships to Other Models of Concurrency.*

[Browne et al. 86] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12), December 1986.

[Bryant & Seger 91] R. E. Bryant and C.-J. H. Seger. Formal hardware verification by symbolic trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[Bryant 86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, August 1986.

[Brzozowski et al. 91] J. A. Brzozowski, T. Gahlinger, and F. Mavaddat. Consistency and satisfiability of waveform timing specifications. *Networks*, January 1991.

[Buck et al. 91] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A platform for heterogeneous simulation and prototyping. *Proceedings of the European Simulation Conference*, June 1991.

[Burch 92] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD dissertation, Carnegie Mellon University, August 1992.

[Burch et al. 90] J. R. Burch, E. M. Clarke, D. L. Dill, and K. L. McMillan. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, June 1990.

[Burch et al. 91] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[Burns 91] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD dissertation, California Institute of Technology, 1991. CS-TR-91-1.

[Chu 87] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* PhD dissertation, Massachusetts Institute of Technology, 1987. MIT/LCS/TR-393.

[Clark et al. 86] E. Clark, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986.

[Clarke et al. 86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986.

[Coen et al. 90] A. Coen, A. Morzentia, and D. Sciuto. Hardware specification with the temporal logic language TRIO. *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[Cohen 90] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7), July 1990.

[Cohen et al. 89] G. Cohen, P. Moller, J. P. Quadrat, and M. Viot. Evaluation of discrete event systems. *Proceedings of the IEEE*, 77(1):39–58, Jan 1989.

[Coolahan & Roussopoulos 85] J. E. Coolahan, Jr. and N. Roussopoulos. A timed Petri net methodology for specifying real-time system timing requirements. In *[TPN 85]*, pages 24–31, 1985.

[Devadas et al. 92] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Verification of asynchronous interface circuits with bounded wire delays. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, November 1992.

[Dill & Clarke 85] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *1985 Chapel Hill Conference on VLSI*, 1985.

[Dill 88] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD dissertation, Carnegie Mellon University, 1988. CMU-CS-88-119.

[Doc 89] Doctor Design, Inc., La Jolla, California. *dV/dt User's Guide*, 1989.

[Doukas & LaPaugh 91] D. Doukas and A. S. LaPaugh. CLOVER: a timing constraints verification system. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[Doukas 91] D. A. Doukas. *A new specification model for timing constraints and efficient methods for their verification.* PhD dissertation, Princeton University, 1991. CS-TR-297-90.

[Ebergen 87] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* PhD dissertation, Technische Universiteit Eindhoven, 1987.

[Eilenberg 74] S. Eilenberg. *Automata, Languages, and Machines, Vol. A.* Academic Press, 1974.

[Emerson & Halpern 86] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1), January 1986.

[Fourman 90] M. P. Fourman. Formal system design. In J. Staunstrup, editor, *Formal Methods for VLSI Design.* North Holland, 1990.

[Fujita et al. 83] M. Fujita, H. Tanaka, and T. Moto-oka. Verification with Prolog and temporal logic. *Proceedings of the 1983 IFIP Conference on Hardware Description Languages (CHDL)*, 1983.

[Fusaoka et al. 84] A. Fusaoka, H. Seki, and K. Takahashi. Description and reasoning of VLSI circuit in temporal logic. *New Generation Computing*, 2, 1984.

[Gahlinger 90] T. Gahlinger. *Coherence and satisfiability of waveform timing specifications.* PhD dissertation, University of Waterloo, 1990. Research Report CS-90-11.

[Garland & Guttag 89] S. J. Garland and J. V. Guttag. An overview of LP: the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications.* Springer-Verlag, 1989.

[Gordon 86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design.* North-Holland, 1986.

[Gordon 88] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Milne and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Publishers, 1988.

[Granacki 86] J. J. J. Granacki. *Understanding Digital System Specifications Written in Natural Language.* PhD dissertation, University of Southern California, December 1986.

[Hansen et al. 92] M. R. Hansen, Z. Chacochen, and J. Staunstrup. A real-time duration semantics for circuits. *Second International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau)*, March 1992.

[Harel 92] D. Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, January 1992.

[Harel et al. 88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. Statemate: A working environment for the development of complex reactive systems. *Proceedings of the 10th IEEE International Conference on Software Engineering*, April 1988.

[Harel et al. 90] E. Harel, O. Lichtenstein, and A. Pneulli. Explicit-clock temporal logic. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[Hayati et al. 88] S. A. Hayati, A. C. Parker, and J. Granacki. Representation of control and timing behavior with applications to interface synthesis. In *IEEE International Conference on Computer Design (ICCD)*, 1988.

[Herstein 64] I. N. Herstein. *Topics in Algebra.* Blaisdell Publishing Company, 1964.

[Hopcroft & Ullman 79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[Hulgaard et al. 93] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. Practical applications of an efficient time separation of events algorithm. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, November 1993.

[Hunt 85] W. A. Hunt, Jr. FM8501: a verified microprocessor. Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.

[Ishiura et al. 89] N. Ishiura, M. Takahashi, and S. Yajima. Time-symbolic simulation for accurate timing verification of asynchronous behavior of logic circuits. In *26th ACM/IEEE Design Automation Conference*, June 1989.

[Ishiura et al. 90] N. Ishiura, H. Yasuura, and S. Yajima. NES: The behavioral model for the formal semantics of a hardware design language UDL/I. In *27th ACM/IEEE Design Automation Conference*, June 1990.

[Jaffar et al. 92] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Jahanian & Mok 86] F. Jahanian and A. K.-L. Mok. Safety analysis of timing properties of real-time systems. *IEEE Transactions on Software Engineering*, September 1986.

[Kara et al. 88] A. Kara, R. Rastogi, and K. Kawamura. An expert system to automate timing design. *IEEE Design and Test of Computers*, pages 28 – 40, Oct 1988.

[Katzenelson & Kurshan 86] J. Katzenelson and R. P. Kurshan. S/R: A language for specifying protocols and other coordinating processes. In *Proceedings of the 5th Annual IEEE International Conference on Computer Communication*, 1986.

[Khordoc et al. 91] K. Khordoc, M. Dufresne, and E. Cerny. A stimulus response system based on hierarchical timing diagrams. *Proceedings of the International Conference on Computer Aided Design*, November 1991.

[Koymans 89] R. Koymans. *Specifying Message Passing and Time Critical Systems with Temporal Logic.* PhD dissertation, Eindhoven University of Technology, 1989.

[Koymans 90] R. Koymans. Specifying real-time properties with metric temporal logic. *Journal of Real-time Systems*, 2, 1990.

[Ku & de Micheli 90] D. Ku and G. de Micheli. HardwareC — A language for hardware design, version 2.0. Technical Report TR CSL–TR-90-419, Computer Systems Laboratory, Stanford University, April 1990.

[Ku 91] D. C.-L. Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications.* PhD dissertation, Stanford University, 1991. CSL-TR-91-476.

[Kurshan & McMillan 91] R. Kurshan and K. L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, November 1991.

[Lai 83] K.-W. Lai. Test program compiler – a high level test program specification language. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 30–31, November 1983.

[Lamport 80] L. Lamport. "sometime" is sometimes "not never". *ACM 7th Principles of Programming Languages*, 1980.

[Lawler 76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[Lazowska et al. 84] E. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queing Network Models.* Prentice Hall, 1984.

[Leeser 89] M. E. Leeser. Reasoning about the function and timing of integrated circuits with interval temporal logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 8(12), December 1989.

[Leeser et al. 91] M. Leeser, A. Takach, and W. Wolf. Behavior FSMs for high-level synthesis and verification. In P. A. Subrahmanyam, editor, *Formal Methods in VLSI Design*. Springer-Verlag, 1991.

[Lewis 90] H. R. Lewis. A logic of concrete time intervals. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[Lynch & Attiya 92] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6, 1992.

[Lynch & Tuttle 89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3), September 1989.

[Malachi & Owicki 81] Y. Malachi and S. S. Owicki. Temporal Specifications of Self-Timed Systems. In H. T. Kung et al., editors, *VLSI Systems and Computations*. Computer Science Press, Rockville MD, 1981.

[Martello & Levitan 93] A. Martello and S. Levitan. Temporal analysis of time bounded digital systems. In *Proceedings of the IFIP WG10.2 Advanced Research Working Conference on Correct Hardware Design Methodologies (CHARME)*, May 1993.

[Martello et al. 90] A. Martello, S. Levitan, and D. Chiarulli. Timing verification using hdtv. In *27th ACM/IEEE Design Automation Conference*, June 1990.

[Martin et al. 89] A. Martin, S. Burns, T. Lee, D. Borković, and P. Hazewindus. The design of an asynchronous microprocessor. In C. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, MA, 1989. MIT Press.

[McFarland 78] M. C. McFarland. The value trace: A database for automated digital design. Technical Report TR DRC-01-04-80, Engineering Design Research Center, Carnegie Mellon University, December 1978.

[McFarland 90] M. C. McFarland. Cpa: Giving an account of timed system behavior. *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[McFarland et al. 90] M. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. In *Proceedings of the IEEE*, volume 78, Feb 1990.

[McGeer & Brayton 91] P. C. McGeer and R. K. Brayton. *Integrating functional and temporal domains in logic design*. Kluwer Academic Publishers, 1991.

[McMillan & Dill 92] K. McMillan and D. L. Dill. Algorithms for interface timing verification. In *IEEE International Conference on Computer Design (ICCD)*, 1992.

[McWilliams 80] T. McWilliams. *Verification of Timing Constraints on Large Digital Systems.* PhD dissertation, Lawrence Livermore Laboratory, May 1980.

[Merlin 74] P. Merlin. *A study of the recoverability of computer systems.* PhD dissertation, University of California, 1974.

[Merritt et al. 91] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In J. Beaten and J. Groote, editors, *Proceedings of CONCUR 91*, volume 257 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[Meyer 85] B. Meyer. On formalism in specification. *IEEE Software*, January 1985.

[Moszkowski 85] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, February 1985.

[Moszkowski 86] B. Moszkowski. Executing temporal logic programs. Technical report, Cambridge University, U.K., 1986.

[Murata 89] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Myers & Meng 92] C. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. In *IEEE International Conference on Computer Design (ICCD)*, September 1992.

[Narain et al. 92] S. Narain, J. Cameron, Y.-J. Lin, and R. C. Sekar. A high-level real-time temporal logic. *Bellcore Internal Publication*, March 1992.

[Nestor 87] J. Nestor. *Specification and Synthesis of Digital Systems with Interfaces.* PhD dissertation, Carnegie-Mellon University, 1987. CMUCAD-87-10.

[Ortega 92] R. Ortega. Operation event timing constraints in Ptolemy. Technical report, Department of Computer Science and Engineering, University of Washington, November 1992.

[Ostroff 90] J. Ostroff. *Temporal Logic of Real-Time Systems.* Research Studies Press, 1990.

[Ousterhout 85] J. Ousterhout. A switch-level timing verifier for digital MOS VLSI. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 4(3), July 1985.

[Park & Shaw 91] C.-Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 25(5):48–57, May 1991.

[Parker & Wallace 81] A. Parker and J. Wallace. SLIDE: An I/O hardware description language. *IEEE Transactions on Computers*, June 1981.

[Pneuli 77] A. Pneuli. The temporal logic of programs. *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[Ramamoorthry & Ho 80] C. Ramamoorthry and G. S. Ho. Performance evaluation of asynchronous conncurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, SE–6:440–449, September 1980.

[Ramchandani 74] C. Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical Report Project MAC TR-120, M.I.T., Cambridge, MA, 1974.

[Rem et al. 83] M. Rem, J. L. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In R. Bryant, editor, *Third CalTech Conference on VLSI*, pages 225–239. Computer Science Press, 1983.

[Rescher & Urquart 71] N. Rescher and A. Urquart. *Temporal Logic*. Springer-Verlag, 1971.

[Sherman 88] S. K. Sherman. Algorithms for timing requirement analysis and generation. In *25th ACM/IEEE Design Automation Conference*, 1988.

[Subramanyam 90] P. A. Subramanyam. Tales: Event-based semantics for timing specification (with applications to synthesis, verification and analysis). *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[Sun & Brodersen 92] J. S. Sun and R. W. Brodersen. Design of system interface modules. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, November 1992.

[TPN 85] *International Workshop on Timed Petri Nets.* IEEE Computer Society Press, July 1985.

[Vahid & Gajski 91] F. Vahid and D. D. Gajski. Obtaining functionally equivalent simulations using VHDL and a time-shift transformation. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 362–365, November 1991.

[van de Snepscheut 85] J. van de Snepscheut. Trace theory and VLSI design. In *Lecture Notes Computer Science 200.* Springer-Verlag, 1985.

[Vanbekbergen et al. 92] P. Vanbekbergen, G. Goossens, and H. D. Man. Specification and analysis of timing constraints in signal transition graphs. In *European Design Automation Conference*, March 1992.

[Wing 90] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, September 1990.

[Wolper 81] P. Wolper. Temporal logic can be more expressive. *Proceedings of the 22nd IEEE Symposium on Foundations of Computer Science*, 1981.

[Zahir & Fichtner 90] R. Zahir and W. Fichtner. Specification of timing constraints for controller synthesis. *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[Zuberek 91] W. M. Zuberek. Timed Petri nets definitions, properties and applications. *Microelectronics and Reliability*, 31(4):627–644, 1991.

Appendix A

The Ethernet Protocol Example

This appendix contains complete details regarding two different versions of a specification of the Ethernet communication protocol described in Chapter 3. The protocol is specified using OEgraphs which describe the behavior of a wire X obeying the protocol.

A.1 Timing Behavior expressed Functionally

```
/* Ethernet, left OEgraph in Figure */
#include "runtime.h"

oe_wire  X("X");
oe_event start("start",X,LOW);
oe_event preamble1("preamble1",X,HIGH);
oe_event preamble_high("preamble_high",X,HIGH);
oe_event preamble_low("preamble_low",X,LOW);
oe_event preamble2("preamble2",X,LOW);
oe_event preamble3("preamble3",X,HIGH);
oe_event data_setup("data_setup",X,VALID);
oe_event data_valid("data_valid",X,VALID);
oe_event finish("finish",X,TRI);

void operation_one(oe_trigger& trigger) {
  int d;
  cout << "=====> enter delay before sending new preamble: "; cin >> d;
  cause(start, d);
}
oe_box one("one", operation_one);

/* The first high transition is at +50 */
void operation_two(oe_trigger& trigger) {cause(preamble1, 50);}
oe_box two("two", operation_two);


/* We loop until the end of the preamble */
void operation_three(oe_trigger& trigger) {
  static int count;
  if (trigger == preamble1) {
    count = 0;
  }
  if (count < 31) {
    cause(preamble_low, 100);
    count++;
  } else {
    cause(preamble2, 50);
  }
}
oe_box three("three", operation_three);

void operation_four(oe_trigger& trigger) {cause(preamble_high, 100);}
oe_box four("four", operation_four);

/* final transition of preamble at +50 */
void operation_five(oe_trigger& trigger) {cause(preamble3, 50);}
oe_box five("five", operation_five);

/* send the data Manchester encoded */
void operation_six(oe_trigger& trigger) {
  static int i, data[100], count;
  if (trigger == preamble3) {
    i = 0;
    cout << "=====> enter data [0,1, or -1 to quit]: "; cin >> data[i++];
    while (data[i-1] == 0 || data[i-1] == 1) {
      cout << "=====> enter data [0,1, or -1 to quit]: "; cin >> data[i++];
    }
    count = 0;
  }
  if (count < i-1) {
    if (data[count++]==0) {
      cause(data_valid,100,LOW);
      if (value_on_port(trigger) == LOW)
cause(data_setup,50,HIGH);
    } else {
      cause(data_valid,100,HIGH);
      if (value_on_port(trigger) == HIGH)
cause(data_setup,50,LOW);
    }
  } else {
    if (value_on_port(trigger) == LOW) {
      cause(data_setup,50,HIGH);
      cause(finish, uniform_delay(350,2050));
    } else {
      cause(finish, uniform_delay(300,2000));
    }
  }
}
oe_box six("six", operation_six);

main(int argc, char* argv[])
{
connect(start, two);
connect(two, preamble1);
connect(preamble1, three);
connect(preamble_high,three);
connect(three, preamble_low);
connect(three, preamble2);
connect(preamble_low,four);
connect(four, preamble_high);
connect(preamble2, five);
connect(five, preamble3);
connect(preamble3, six);
connect(data_valid,six);
connect(six, data_valid);
connect(six, data_setup);
connect(six, finish);
connect(finish,one);
connect(one,start);

oesim(argv[0], options(argc, argv));
}
```

A.2 Timing Behavior expressed using Timing Constraints

```
/* Ethernet, right OEgraph in Figure 11 */
#include "runtime.h"

#define PREAMBLE_LENGTH 64

oe_wire  X("X");
oe_event X_valid("X_valid",X,VALID);
oe_event finish("finish",X,TRI);

void operation_one(oe_trigger& trigger) {
  static int count, limit;
  if (trigger == finish) {
    count = 0;
    limit = unspecified_integer();
  }
  if (count < limit) {
    cause(X_valid, unspecified_integer(), unspecified_value());
    count++;
  } else {
    cause(finish, unspecified_integer());
  }
}
oe_box one("one", operation_one);

main(int argc, char* argv[])
{
connect(one, finish);
connect(one, X_valid);
connect(finish,one);
connect(X_valid,one);

oe_set X_HIGH(*(new set_node(X)) ^ *(new set_node(SET_HIGH)));
oe_set X_LOW( *(new set_node(X)) ^ *(new set_node(SET_LOW)));
oe_set X_TRI( *(new set_node(X)) ^ *(new set_node(SET_TRI)));

discrete_name x1("x1",X);
discrete_name x2("x2",X);
discrete_name xL("xL",X_LOW);
discrete_name xTRI("xTRI",X_TRI);
discrete_name xTRI2("xTRI2",X_TRI);

oe_constraint c1("c1");
c1.quantify(xTRI,x1,x2);
c1.context(nco(xTRI, X, x1) &
   nco(x1, X, x2));
c1.require(valueOf(x1)==LOW &
   valueOf(x2)==HIGH &
   timeOf(x2) - timeOf(x1) == 50);
c1.error("1st transition after tri should be low then high at +50ns");

oe_constraint c2("c2");
c2.quantify(xTRI, xL, x1, x2);
c2.context(nco(xTRI, X_LOW, xL) &
   timeOf(x2)-timeOf(xL) <= 50+(PREAMBLE_LENGTH-2)*100 &
   timeOf(x2)-timeOf(xL) > 50 &
   nco(x1, X, x2));
c2.require((timeOf(x2)-timeOf(xL)) % 100 == 50 &
   ( (valueOf(x1)==HIGH & valueOf(x2)==LOW) |
     (valueOf(x1)==LOW & valueOf(x2)==HIGH)));
c2.error("preamble error, high/low from 50 (high) to 11 every 100");


/* alternatively, constraint c2 (which is very complicated) could be
   expressed as a large number of simpler constraints */

for(int i=2; i < PREAMBLE_LENGTH; i++) {

  oe_constraint* preamble_constraint =
    new oe_constraint ("preamble_constraint");
  preamble_constraint->quantify(xTRI,xL,x1);
  preamble_constraint->context(nco(xTRI,X_LOW,xL) &
       nco(xL,i,X,x1));
  preamble_constraint->require(timeOf(x1)-timeOf(xL)==(i-1)*100+50 &
       valueOf(x1) ==
         ((i % 2 == 0)? LOW : HIGH));
}
*/

oe_constraint c3("c3");
c3.quantify(xTRI,xL,x1);
c3.context(nco(xTRI,X_LOW,xL) &
   nco(xL,PREAMBLE_LENGTH,X,x1));
c3.require(timeOf(x1)-timeOf(xL)==(PREAMBLE_LENGTH-1)*100 &
   valueOf(x1)==LOW);
c3.error("end of preamble invalid");

oe_constraint c4("c4");
c4.quantify(xTRI,xL,x1);
c4.context(nco(xTRI,X_LOW,xL) &
   nco(xL,PREAMBLE_LENGTH+1,X,x1));
c4.require(timeOf(x1)-timeOf(xL)==(PREAMBLE_LENGTH-1)*100 + 50 &
   valueOf(x1)==HIGH);
c4.error("end of preamble invalid");

oe_constraint c5("c5");
c5.quantify(xTRI,xL,x1);
c5.context(nco(xTRI,X_LOW,xL) &
   nco(xL,PREAMBLE_LENGTH+2,X,x1));
c5.require(valueOf(x1)==TRI |
   (valueOf(x1)==LOW &
    (timeOf(x1)-timeOf(xL)==PREAMBLE_LENGTH*100 |
     timeOf(x1)-timeOf(xL)==PREAMBLE_LENGTH*100+50)));
c5.error("problem with first data transmitted");

oe_constraint c6("c6");
c6.quantify(xTRI,xL,x1,x2,xTRI2);
```

```
c6.context(nco(xTRI,X_LOW,xL) &
   nco(xTRI,X_TRI,xTRI2) &
   timeOf(x2)-timeOf(xL) >= PREAMBLE_LENGTH*100+50 &
   timeOf(x2) < timeOf(xTRI2) &
   nco(x1,X,x2));
c6.require(((valueOf(x1)==LOW & valueOf(x2)==HIGH) |
    (valueOf(x2)==LOW & valueOf(x1)==HIGH)) &
   ((timeOf(x2)-timeOf(xL)) % 100 == 50 &
    (timeOf(x2)-timeOf(x1) == 50 |
     timeOf(x2)-timeOf(x1) == 100)) |
   ((timeOf(x2)-timeOf(xL)) % 100 == 0 &
    (timeOf(x2)-timeOf(x1) == 50)));
c6.error("Manchester encoding of data is faulty");

oe_constraint c7("c7");
c7.quantify(xTRI, x1);
c7.context(pco(xTRI, X, x1));
c7.require(valueOf(x1)==HIGH &
   timeOf(xTRI)-timeOf(x1) >= 300 &
   timeOf(xTRI)-timeOf(x1) <= 2000);
c7.error("problem with protocol termination");

oesim(argv[0], options(argc, argv));
}
```

Appendix B

The SN74LS222 Example

This appendix contains complete details regarding the Texas Instruments 16 element FIFO storage queue used as an example in Chapter 4.

B.1 Specification of the LS222

The complete OEgraph specification for the LS222 as a library component:

```
/* simulation model for SN74LS222 */

#include "rt_support.h"  // include the run time support library for OEsim
int max_size = 4;  // actual (# FIFO entries) is 16, use 4 for convenience

/* define a new encapsulation class, the LS222 */

class operation_sn74ls222: public s_bbox {
        oe_wire *not_CLR,*LD,*UN,*IR,*OR,*D,*Q;
        int     amount;
        OEqueue memory;
        wire_value prev_UN,prev_LD;
public:

/* every encapsulation has a constructor (whose parameters are the
   inputs and outputs of the encapsulated operation) and an evaluate
   routine which is activated whenever an input event occurs */

        operation_sn74ls222(char *name,
           oe_wire &a_not_CLR, oe_wire &a_LD, oe_wire &a_UN,
           oe_wire &a_IR, oe_wire &a_OR, oe_wire &a_D, oe_wire &a_Q);
        void evaluate(oe_trigger&);
};
```

```
/* we make use of a constraint subroutine to simplify constraint definition */

pulse_min(oe_event_or_wire &A, wire_value& valA, oe_event_or_wire &B,
          wire_value& valB, int time)
{
  oe_constraint* pulse_min = new oe_constraint ("pulse_min");

  /* the discrete names are created if necessary */
  discrete_name &A0 = find_or_make_dname(A, "0", valA);
  discrete_name &B1 = find_or_make_dname(B, "1", valB);

  /* the constraint is specified using the restricted event-logic */
  pulse_min->quantify(A0, B1);
  pulse_min->context(nco(A0, 1, *(B.my_set_of(&valB)), B1));
  pulse_min->require(timeOf(B1)-timeOf(A0) >= time);
}

/* the constructor,  used when one instantiates an LS222

operation_sn74ls222::operation_sn74ls222(char *name,
  oe_wire &a_not_CLR, oe_wire &a_LD, oe_wire &a_UN,
  oe_wire &a_IR, oe_wire &a_OR, oe_wire &a_D, oe_wire &a_Q) : (name)
{
  /* the actual connections are stored for use during evaluation */
  not_CLR = &a_not_CLR; LD = &a_LD; UN = &a_UN; IR = &a_IR;
  OR = &a_OR; D = &a_D; Q = &a_Q;

  /* the actual inputs and outputs are connected to this encapsulation */
  connect(a_not_CLR, *this); connect(a_LD, *this); connect(a_UN, *this);
  connect(*this, a_IR);      connect(*this, a_OR);
  connect(a_D, *this);       connect(*this, a_Q);

  /* the timing constraints from the databook are specified */
  pulse_min(a_LD,HIGH,a_LD,LOW,60);
  pulse_min(a_LD,LOW,a_LD,HIGH,15);
  pulse_min(a_UN,LOW,a_UN,HIGH,30);
  pulse_min(a_UN,HIGH,a_UN,LOW,30);
  pulse_min(a_not_CLR, LOW, a_not_CLR, HIGH, 20);
  pulse_min(a_LD,LOW,a_UN,LOW,50);
  pulse_min(a_UN,HIGH,a_LD,HIGH,50);
  pulse_min(a_D,VALID,a_LD,LOW,50);

  prev_UN = UNDEFINED;
  prev_LD = UNDEFINED;
}

/* the evaluate routine is defined.  The code is quite similar to that of
   a non-encapsulated specification, except that the inputs are
   pointers to wires, and not actually wires. */

void operation_sn74ls222::evaluate(oe_trigger &trigger) {
  OEint tmp;

  if (trigger == *not_CLR && *not_CLR == LOW) {
    cause(*IR,uniform_delay(36,55),HIGH);
    cause(*OR,uniform_delay(25,40),LOW);
    amount = 0;
  }

  if (*not_CLR == LOW) {   // async reset, capture possible data changes
    prev_UN = *UN;
    prev_LD = *LD;
    return;
  }

  if (trigger == *LD) {
    if (*LD==LOW) {
      if (prev_LD == HIGH && amount < max_size) { //edge triggered
        if (amount == 0) {  //put data onto Q instead of memory
          amount = 1;
          cause(*Q,uniform_delay(34,50),value_on_port(*D));
          if (*UN==HIGH) cause(*OR,uniform_delay(48,70),HIGH); //not empty
        } else {
          amount++;
          memory.enqueue(*D,ancestors(*D,*LD));
        }
      }
      if (amount != max_size) cause (*IR,uniform_delay(25,40), HIGH);
    }
    if (*LD==HIGH) {
      cause(*IR,uniform_delay(36,50),LOW);
    }
    prev_LD = *LD;
  }

  if (trigger == *UN) {
    if (prev_UN == LOW && *UN==HIGH && amount > 0) { //edge triggered
      if (amount ==1) {
        amount--;
        cause(*Q,uniform_delay(54,80),TRI);
        cause(*OR, uniform_delay(48,70), LOW);   //since empty
      } else { //Q gets memory when amount > 1
        amount--;
        tmp = memory.dequeue();
        cause(*Q,uniform_delay(45,70),tmp,ancestors(tmp));
        if (*LD==LOW) cause(*IR,uniform_delay(49,70),HIGH); //not full
      }
    }
    if (*UN==HIGH && amount > 0) cause(*OR, uniform_delay(29,45), HIGH);
    if (*UN==LOW) cause(*OR,uniform_delay(28,45),LOW);
    prev_UN = *UN;
  }
}
```

B.2 Simulation of the LS222

We simulate two LS222s which have been composed together (see Figure 4.6). For this simulation, we assume that each FIFO holds 4 integers of data, and we issue a series of 7 load commands (loading 10, 20, 30, 40, 50, 60, 70) followed by 7 unload commands. Here is the simulation log:

```
Welcome To Simulation v1.3, Mon Aug 13 11:41:00 1990
... reading stimulus file LS222.itf
oesim-0> run
sim_event_occurs at time:   0 event not_CLR$<external> (not_CLR*LOW)
sim_event_occurs at time:   28 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   31 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   44 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   51 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   200 event not_CLR$<external> (not_CLR*HIGH)
sim_event_occurs at time:   1000 event LD1$<external> (LD1*HIGH)          the first load
sim_event_occurs at time:   1000 event D1$<external> (D1*10)
sim_event_occurs at time:   1046 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   1100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   1135 event Q1_D2$fifo1 (Q1_D2*10)
sim_event_occurs at time:   1135 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   1149 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   1187 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   1221 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   1259 event Q2$fifo2 (Q2*10)
sim_event_occurs at time:   1260 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   1321 event Q1_D2$fifo1 (Q1_D2*TRI)
sim_event_occurs at time:   1324 event O1L2$fifo1
sim_event_occurs at time:   2000 event LD1$<external> (LD1*HIGH)          the second load
sim_event_occurs at time:   2000 event D1$<external> (D1*20)
sim_event_occurs at time:   2045 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   2100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   2134 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   2135 event Q1_D2$fifo1 (Q1_D2*20)
sim_event_occurs at time:   2168 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   2218 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   2262 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   2295 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   2350 event O1L2$fifo1
sim_event_occurs at time:   2369 event Q1_D2$fifo1 (Q1_D2*TRI)
sim_event_occurs at time:   3000 event LD1$<external> (LD1*HIGH)          the third load
sim_event_occurs at time:   3000 event D1$<external> (D1*30)
sim_event_occurs at time:   3036 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   3100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   3135 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   3147 event Q1_D2$fifo1 (Q1_D2*30)
sim_event_occurs at time:   3159 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   3200 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   3232 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   3264 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   3333 event O1L2$fifo1
sim_event_occurs at time:   3342 event Q1_D2$fifo1 (Q1_D2*TRI)
sim_event_occurs at time:   4000 event LD1$<external> (LD1*HIGH)          the fourth load
sim_event_occurs at time:   4000 event D1$<external> (D1*40)
sim_event_occurs at time:   4043 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   4100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   4139 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   4143 event Q1_D2$fifo1 (Q1_D2*40)
sim_event_occurs at time:   4160 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   4210 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   4249 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   5000 event LD1$<external> (LD1*HIGH)          the fifth load
sim_event_occurs at time:   5000 event D1$<external> (D1*50)
sim_event_occurs at time:   5042 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   5100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   5129 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   6000 event LD1$<external> (LD1*HIGH)          the sixth load
sim_event_occurs at time:   6000 event D1$<external> (D1*60)
sim_event_occurs at time:   6039 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   6100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   6139 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   7000 event LD1$<external> (LD1*HIGH)          the seventh load
sim_event_occurs at time:   7000 event D1$<external> (D1*70)
sim_event_occurs at time:   7044 event IR1$fifo1 (IR1*LOW)
sim_event_occurs at time:   7100 event LD1$<external> (LD1*LOW)
sim_event_occurs at time:   31000 event UN2$<external> (UN2*LOW)          the first unload
sim_event_occurs at time:   31041 event OR2$fifo2
sim_event_occurs at time:   31100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   31131 event OR2$fifo2 (OR2*HIGH)
sim_event_occurs at time:   31152 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   31169 event Q2$fifo2 (Q2*20)
sim_event_occurs at time:   31195 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   31198 event Q1_D2$fifo1 (Q1_D2*50)
sim_event_occurs at time:   31203 event IR1$fifo1 (IR1*HIGH)
sim_event_occurs at time:   31232 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   31264 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   32000 event UN2$<external> (UN2*LOW)          the second unload
sim_event_occurs at time:   32032 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   32100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   32134 event OR2$fifo2 (OR2*HIGH)
sim_event_occurs at time:   32161 event Q2$fifo2 (Q2*30)
sim_event_occurs at time:   32164 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   32196 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   32230 event Q1_D2$fifo1 (Q1_D2*60)
sim_event_occurs at time:   32232 event IR1$fifo1
sim_event_occurs at time:   32239 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   32281 event O1L2$fifo1 (O1L2*LOW)
sim_event_occurs at time:   33000 event UN2$<external> (UN2*LOW)          the third unload
sim_event_occurs at time:   33039 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   33100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   33129 event OR2$fifo2 (OR2*HIGH)
sim_event_occurs at time:   33156 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   33159 event Q2$fifo2 (Q2*40)
sim_event_occurs at time:   33185 event O1L2$fifo1 (O1L2*HIGH)
sim_event_occurs at time:   33214 event IR1$fifo1
sim_event_occurs at time:   33216 event Q1_D2$fifo1 (Q1_D2*70)
sim_event_occurs at time:   33230 event U1I2$fifo2 (U1I2*LOW)
sim_event_occurs at time:   33263 event O1L2$fifo1 (O1L2*LOW)

 ** Constraint Violation: pulse_min:   Q1_D2-VALID0 O1L2-LOW1 :
 nco(1, Q1_D2-VALID0, O1L2-LOW1) ==> ((t(O1L2-LOW1) - t(Q1_D2-VALID0)) >= 50)
 Q1_D2-VALID0 =  unique event: Q1_D2$fifo1 occurrence: 10 at time: 33216
 O1L2-LOW1 =  unique event: O1L2$fifo1 occurrence: 18 at time: 33263

sim_event_occurs at time:   34000 event UN2$<external> (UN2*LOW)          the fourth unload
sim_event_occurs at time:   34042 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   34100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   34144 event OR2$fifo2 (OR2*HIGH)
sim_event_occurs at time:   34167 event Q2$fifo2 (Q2*50)
sim_event_occurs at time:   34170 event U1I2$fifo2 (U1I2*HIGH)
sim_event_occurs at time:   34227 event Q1_D2$fifo1 (Q1_D2*TRI)
sim_event_occurs at time:   34237 event O1L2$fifo1
sim_event_occurs at time:   35000 event UN2$<external> (UN2*LOW)           the fifth unload
sim_event_occurs at time:   35028 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   35100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   35133 event OR2$fifo2 (OR2*HIGH)
sim_event_occurs at time:   35161 event Q2$fifo2 (Q2*60)
sim_event_occurs at time:   35167 event U1I2$fifo2
sim_event_occurs at time:   36000 event UN2$<external> (UN2*LOW)           the sixth unload
sim_event_occurs at time:   36038 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   36100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   36141 event OR2$fifo2 (OR2*HIGH)
sim_event_occurs at time:   36151 event U1I2$fifo2
sim_event_occurs at time:   36169 event Q2$fifo2 (Q2*70)
sim_event_occurs at time:   37000 event UN2$<external> (UN2*LOW)         the seventh unload
sim_event_occurs at time:   37037 event OR2$fifo2 (OR2*LOW)
sim_event_occurs at time:   37100 event UN2$<external> (UN2*HIGH)
sim_event_occurs at time:   37161 event Q2$fifo2 (Q2*TRI)
sim_event_occurs at time:   37170 event OR2$fifo2
 stopped at time: 99999
oesim-99999>
```

Vita

Tod Amon was born on April 28, 1963 and grew up in Colorado. His undergraduate work was done at the University of Colorado in Boulder, from where he graduated in May, 1985 with a Bachelor of Science degree in Applied Mathematics with a minor in Computer Science. Prior to completion of his degree, he worked for the Laboratory for Atmospheric and Space Physics and was involved with the Solar Mesosphere Explorer, a NASA satellite operated by the university. He worked as a Senior Software Engineer for General Dynamics in San Diego from June of 1985 until June of 1988. He was a Project Manager in the CAD/CAM department and worked on factory scheduling and simulation software. He attended the University of Washington from September of 1988 until the completion of this dissertation in July of 1993, working as a Research and Teaching Assistant except for one year during which he held an IBM Graduate Fellowship. He joined Southwest Texas State University in September of 1993.