Chapter 1

Introduction

'Where shall I begin, please your Majesty?' he asked.
'Begin at the beginning,' the King said, gravely, 'and go on till
you come to the end: then stop.' —Lewis Carroll, 1832–1898

The development of integrated digital electronic circuits—microchips—is the most astounding and important engineering achievement of the twentieth century. As circuit speeds and densities increase at exponential rates, computer-aided design tools take over increasing amounts of the design process. The limits on what the tools can do, rather than the limits of human ingenuity, are becoming the constraints on what chips can be designed.

Meanwhile, competitive pressures are pushing us away from old abstractions and design styles that made design much easier at the expense of time and area. New design styles are rapidly evolving to fill the gap, introducing new assumptions and requirements on the design process.

In the midst of all of this progress, some basic questions remain open. What is an effective specification language for circuits that does not imply a particular design style or prematurely constrain a circuit design? What formalism can be used to adequately model circuit behavior for simulation, describe specifications for synthesis, and compare specifications to implementations for verification? This dissertation is an attempt to design one such specification language and representation. The computer program instantiating these ideas is called *Orbits*, and the representation itself is called *orbital nets*.

1.1 Issues in Hardware Design

In order to understand why so much attention has been given to hardware design, we examine how styles of hardware design have evolved over the years. In this section, we provide some background and discuss this evolution.

Hardware design is difficult for two major reasons. The first is the sheer size and complexity of most modern circuits; chips with millions of gates are common. The second is the wide range of disciplines spanned by circuit engineering. Materials science and field theory describe low-level behavior of transistors and other components. These transistors are combined to form gates, and Boolean logic is used to reason about the connections between these gates. Both control logic and data paths are constructed out of gates. Automata theory is used to describe and reason about control logic. The controllers often instantiate significant programs, so programming skills are required. Finally, circuits tend to have many portions operating in parallel, so a knowledge of issues in concurrent systems is important.

At the lowest level, the electrical current flow through a transistor is controlled by the voltage present on its terminal wires. By combining transistors in certain ways, circuits can be built that generate output voltages related in a specific way to the input voltages applied; whether a voltage is high or low can indicate a single truth value, or bit, of information. Once it has been shown that a particular collection of transistors performs the Boolean not-and function on its inputs, that collection of transistors can be replicated anywhere the Boolean function is required. Any complex Boolean function of many inputs can be decomposed into a collection of these simple not-and functions, so any desired Boolean function can be created. We can also perform arithmetic, since arithmetic operations on integers can be reduced to Boolean operations on vectors of bits. If the gates are connected without feedback—that is, the graph formed by the interconnected gates is acyclic—then the gates obey simple arithmetic and Boolean laws, and analysis is straightforward. Such an acyclic circuit is called combinational logic.

But gates connected in such a way cannot maintain state—there is no way for such a circuit to remember anything. Feedback loops and capacitance are therefore generally used to introduce state. Unfortunately, no simple theory such as Boolean logic exists that describes the behavior of circuits with feedback loops. Controlling the feedback loops so that the circuit functions correctly and at a pace with its environment is of fundamental importance in circuit design.

The most successful approach, and indeed the approach still dominant today, breaks all feedback loops with registers or latches, which are circuit elements that store values presented on their inputs when they receive a particular signal. This signal is typically a change from low voltage to high voltage (a rising edge) on a clock wire shared among all such registers or latches. These registers act as barriers that divide the circuit into acyclic sections, each of which is easily analyzed and constructed. The global clock manages the control of data through the registers, and through them, the feedback loops. The state of the circuit only changes on these clock edges; no matter how many parts of the circuit there are, all state transitions occur on clock edges in synchronization with the rest of the circuit. For this reason, this design style is called synchronous design.

The reader may have seen advertisements for a 50 MHz 486 computer. The 50 MHz represents the rate at which the clock changes, in this case fifty million times a second. This clock rate determines the maximum number of state changes, and thus operations, the circuit can perform in a second. There is a limit to how fast the clock can be. The transistors that comprise the circuit can only switch at a certain speed, and can only conduct a finite amount of current, so each gate will have some amount of delay. In order for the circuit to perform correctly, the time between clock edges must exceed the maximum delay of any combinational logic component of the circuit. In other words, the maximum speed of the circuit is set by the slowest section of combinational logic. In addition, because of variations in the supply voltage, temperature, and the fabrication of the chip, the clock can be no faster than the worst-case delay of the slowest combinational logic element. So while these timing issues are important in synchronous design, they can be reduced to a few simple requirements dominated by an upper limit on the clock speed.

The synchronous design style has some additional problems. As chip sizes increase, it becomes more difficult to propagate the global clock signal around the chip so that each register receives a clock edge at the same time. This delay—the maximum difference in time that the same clock edge arrives at different portions of the circuit—further lengthens the minimum clock period. Another problem is metastability. If the value presented to a latch is changing at the same time a clock edge occurs, the output of that latch might be either high or low, or some intermediate value, or it might even oscillate between high and low for an indeterminate amount of time. This behavior can lead to circuit failures. Any inputs not controlled by the clock can occur at any time and thus subject the circuit to such failures; the design must add additional logic and delay to compensate for such occurrences.

These problems, and others, have increased interest in other design styles. These new styles do not depend on a global clock for proper operation; they are collectively called asynchronous design styles. We discuss the different types of asynchronous design in more depth when we consider the relevant existing research. For some of these design styles, timing is much more critical and complex than in synchronous design. Completely new algorithms and design tools have therefore become necessary.

1.2 Computer-Aided Design Tools

Because synchronous design has dominated the industry, most computer-aided design systems assume this type of design. But even for synchronous designs, computer-aided design tools have traditionally specialized in assisting the designer at one specific level of design. A synthesis package takes a high-level specification and generates a circuit; the designer can neither incorporate low-level timing information or partial architecture into a high-level specification, nor participate in the design process once the system takes over. Various logic synthesis tools exist for minimizing the size or delay of a set of Boolean equations that must be implemented, but these are seldom useful for data path synthesis. Design capture and simulation tools at the layout level can help a designer build a circuit rectangle by rectangle and verify that it works, but the designer must already have a register-transfer level design to work from.

One of the greatest problems with all of these tools is that they each use their own representation of the design, and the representations they use cannot represent design information not directly relevant to the task they perform. While this makes the tools simpler, it makes it much more difficult to use the tools in concert to build a more comprehensive design system.

As tools take on more and more of the design tasks, two things happen. First, designers are distanced from the design process: once a high-level description of the design is written, the tools take over. Thus, the designer's extensive design knowledge gained through long experience cannot be used to help build a faster, smaller circuit [8]. Second, in order for the tools to take on the complex design task, simplifying assumptions must be made. For instance, one tool might synthesize all designs to a two-bus data path with a register file for storage. But the major simplifying assumption that most synthesis tools make is that synthesis can be done in much the same way that traditional software compilers translate Pascal to assembly. This assumption is almost universally reflected in the input languages to the synthesis tools, which most often resemble a programming language for a sequential computer.

Unfortunately, hardware design and software compilation are radically different tasks. Hardware is by nature parallel; in general, the more parallelism in a particular design, the more efficient the design is. Sequential programming languages, on the other hand, are based on a single control flow through subroutines and loops; there is generally no simple way to specify or automatically extract potential parallelism. Second, the potential design space of hardware is much larger than the design space of software. Compiling to assembly language essentially chooses an order for a small set of instructions; the various solutions are usually similar in terms of performance. The hardware designer, on the other hand, must not only select the appropriate functional units, but must also design a bus architecture and a timing and control strategy, and the alternatives may differ in performance by an order of magnitude. Finally, inefficiency in hardware design brings a much greater penalty than it does in software compilation. If a program runs 50% slower and is 200% larger than the best possible program, it is almost certainly still acceptable. A circuit design that is three times as big as it needs to be, however, is almost certainly unacceptable; increasing the die size brings a large decrease in yield and a huge increase in cost.

1.3 Objectives of a Representation

For some circuits, the inefficiencies introduced by the current generation of compiler-based automatic compilation tools are more than offset by the decreased time to market. And, for certain designs, current automatic synthesis tools can do as well as, and in some cases better than, a human designer. But for the large set of remaining designs, more powerful design tools are still needed. Some key requirements for these tools are:

- A specification language that does not prematurely constrain the design. A circuit specification should not depend on whether it is implemented with synchronous design or asynchronous design.
- A representation rich enough to include data flow, control flow, and timing, as well as the complex relationships among them, in models and specifications.
- New or adapted algorithms for synthesis, simulation, and verification that operate on this representation.

All three of these items directly relate to the internal representation of design information. In this section we outline the objectives of our design representation. We illustrate that we need to both model the behavior of circuit elements and specify the allowable behavior of the final circuit. We compare the different features a representation must have for these two different tasks, and consider the features our representation must support to be useful in a design tool.

1.3.1 The representation as the language of design

The purpose of a design representation is to enable the computer (and often, the designer) to answer questions about a particular design. The representation provides an abstract mathematical world with a few simple objects and operations that can be used for modeling and reasoning about the behavior of circuits.

The normal operations involved in design generally accept their inputs and generate their outputs in terms of this representation. Simulation takes a circuit structure or specification and returns a sample behavior of that circuit. Such a behavior is generally given as a sequence of events. To the computer, these events are just labels; to the human designer, they represent physical occurrences in the circuit. Verification takes a circuit and a specification and either proves that the circuit meets the specification or provides a sample behavior of the circuit that is not allowed by the specification or the circuit element models. Synthesis takes a specification and returns a circuit. The operations themselves might proceed as transformations on the representation, or they might proceed in some other manner; but in general, communication of design information into and out of these design operations is given specifically in the design representation.

A simple representation will often lead to simple mathematical operations and rich theories and algorithms. But a simple representation might not be as accurate in modeling the circuits, or it might limit the accuracy of a specification. It might make specification of certain information difficult, leading to a large model for what is an uncomplicated piece of design information. This, in turn, might require the computer to perform an excessive number of operations to answer a simple question, or it might make it impossible for the designer to specify some crucial design parameter, thus rendering it useless for that design task.

A more complex representation might accurately represent more design information (although this is not guaranteed; witness the many complex variants of Pascal that add little expressiveness), but its complexity might inhibit the development of theories or analysis techniques. With a complex representation, the interactions between different parts of the representation might introduce subtle flaws in algorithms, circuit models or specifications.

A balance must be struck between a simple but well understood representation and a more complex, richer representation. As we develop our representation throughout this dissertation, we will discuss each additional feature and explain how it carries its weight.

1.3.2 Describing circuit behavior

Both modeling and specification describe circuit behavior. Generally, a model describes how a physical circuit element responds to particular stimuli, and a specification prescribes the required behavior, the set of inputs that must be handled, and certain properties of the circuit. The former is generally operational while the latter is usually declarative.

Simulation and verification both require accurate models. Synthesis techniques assume an implicit library of components and their associated behavior, but some advanced tools are able to derive information from libraries of primitives given in some representation.

Modeling circuit behavior generally requires some sort of interpretable specification that provides the response of the circuit to a set of stimuli. These models must be composable in order to yield executable models of composite circuits.

Constructing circuit models is itself a nontrivial task. Most circuits are designed to work properly only when the input meets certain constraints; for instance, a flip-flop has specific setup and hold times that, if violated, can cause the circuit to go metastable. The circuit models must be able to ensure that these constraints on the input are satisfied.

In addition, in many circuits, temporarily unused portions of the circuit may have their operating conditions violated. For instance, many fully functional circuits have non-fatal hazards or race conditions. (A hazard is a spurious signal, such as an input to an inverter that rises and falls quickly, possibly more quickly than the gate can respond.) The circuit itself will fail only if these hazards can propagate to the outputs of the circuit—and sometimes not even then. Thus, circuit models often must include some conservative estimate of how a circuit will respond to adverse inputs and how the circuit can exit failure mode.

The circuit models must accurately model unknown or variable parameters or non-deterministic choices made by the physical circuit. Delays should be given as a range or distribution rather than a unique value, because the delay of a circuit can vary under different conditions. For formal verification, the model must enable us to obtain the full set of possible responses to a given set of stimuli, rather than just one arbitrary response.

1.3.3 Prescribing required behavior

In addition to physical circuit behavior, the representation must allow the designer to specify the desired behavior of the final design, usually as an abstract specification that gives the set of possible inputs and permissible behaviors. This specification can be compared with a potential implementation by exploring the reachable state graph of the combined system.

When used in this way as a specification, different demands are placed on the representation than when used for modeling. When modeling, additional detail is often not harmful. A delay element that allows hazards will suffice as a model even if the composite circuit is hazard-free. A specification, on the other hand, delineates all permissible behaviors—and does not provide any unnecessary structural or other design information. As an example, if the internal clocking paradigm of the circuit is irrelevant, then either an asynchronous self-timed circuit or a synchronous clocked circuit would satisfy the same specification.

It is precisely this observation that explains why VHDL and some other simulation languages are such poor synthesis languages—in order to provide a specification that includes sufficient detail to be useful, much of the structure of the circuit must also be specified. As a further motivation, consider timing constraints. A circuit model generally includes timing constraints as the set of all possible delays that a circuit can generate—and its environment had better be prepared to cope with the full range of delays. A timing constraint in a specification, on the other hand, is a set of permissible delays—any individual delay or subset of this range is permitted. These two different ideas are not always distinguished in specification languages or representations.

Generally, a specification language need not be interpretable, but it must allow particular behaviors to be compared against specifications. There are many simple specifications that involve timing that are not easily interpretable—it is possible for an interpreter to paint itself into a corner and be required to backtrack arbitrarily far. For a simple analogy, consider circuit modeling as conventional programming and specification as constraint programming.

1.3.4 Advantages to a common formalism

Despite the differences between modeling and specification, there is an important duality between specifications and models. A specification, by describing the set of inputs that the circuit must accept, can function as a model of the environment of that circuit. The specification can supply the stimuli for simulation runs; this requires that the specification be operational rather than declarative.

There are advantages to using the same formalisms for a specification and for an implementation. Techniques or tools that only handle a single formalism can be applied to design tasks. Also, hierarchical verification can be performed, which helps address the state explosion problem.

1.3.5 Types of design information

Circuit design involves many different types of design information. This dissertation considers three categories: control, data flow, and timing. We define *control* as the required ordering of events on the input and output wires of a circuit. Control includes the outputs that must be generated as well as the inputs that must be handled.

A great many design decisions and much design information are concerned with *timing*. When all we have is an abstract specification, the timing information is usually limited to a description of the input and output signals to the circuit. This includes the signaling discipline used to load inputs and generate outputs, the maximum delay from an input to an output, and the set of legal inputs. As the design proceeds, more timing issues arise. Data flow operations must be scheduled. Values must be assigned to wires for specific durations. The timing requirements of library components used to construct the circuit must be taken into account. Our representation must be able to describe all of these timing issues, from the initial specification to the final circuit.

The specific values a circuit computes from its inputs are often the primary motivating force behind a design. Thus, our representation must be able to handle abstract values and computations, or *data flow*. At the high level, we must not impose arbitrary orderings on portions of the computation, and we must allow such transformations as partial evaluation, loop unrolling or expanding, and all the usual compiler optimizations on the data flow of our circuit.

The final measure of the quality of our representation is what we can do with it. We intend to show that our representation yields a concise and usable specification of both design requirements and circuit behavior. We will show how several existing design algorithms can be adapted to work with our representation.

1.4 Contributions

Our contributions span the areas of specification languages, representations, and algorithms. We introduce a specification language that is not designed around just simulation or synthesis. Instead, it describes circuit behavior, allowing as much design freedom as desired. The semantics of the language is given by the semantics of the orbital nets that result from evaluation of the language. We illustrate this language and its underlying representation by constructing and using some example models and specifications throughout the dissertation.

We also provide a rich and powerful representation that supports data flow, control flow, and timing. The representation is a computationally efficient representation of trace structure semantics for control and timing, allowing for the first time hierarchical timed trace structure verification.

Finally we contribute new, efficient timing algorithms. We prove that, for orbital nets and many other forms of timed automata, a rich class of behavioral properties yields the same results under a discrete time model and a continuous time model. This result applies in particular for orbital nets and trace structure conformance, and in general for all firing sequence properties of time Petri nets. We provide a complexity improvement for the basic transition firing operation in geometric timing algorithms from $O(n^3)$ to $O(n^2)$, and we introduce a sequence of improvements to reachability graph exploration that can decrease verification times by orders of magnitude.

1.5 Overview

We have divided our presentation into ten chapters. Each chapter builds on the material presented in the previous chapters, adding more concepts, operations, or algorithms. The introductory chapter discusses the problem being solved and how Orbits relates to previous research. Chapter two introduces the design representation. The Petri net formalism that forms the base for orbital nets is described. We illustrate the base representation with models of circuit elements, and we compare these models with other, similar models that have been used before.

Chapter three discusses the need for a textual language for the representation and presents the basics of this language. We describe the basic conflict in building such a language: using straightforward Petri net constructions for basic primitives such as parallel composition and choice introduces undesired unlabeled Petri net transitions. We discuss how the language is evaluated to form orbital nets and describe the role of partial evaluation in this process for constructing state machines and propagating the initial values of wires.

Chapter four continues the development of the theory behind our representation, giving a behavioral semantics and structural constructions for receptiveness and composition, and showing that orbital nets form a trace structure algebra and thus also a circuit algebra. The textual language is extended to handle these operations. We discuss the need for sets of labels on transitions and demonstrate how we work around the difficulties introduced by this. Finally, we prove some computationally useful properties of orbital nets.

Chapter five explains trace structures in more depth, especially the operations of mirroring and conformation. We discuss how other possible behavioral semantics do not reflect the intuitive idea of conformance for circuits. We show how trace structure operations apply to orbital nets, and we present simulation and verification through state space exploration. We then present a number of examples illustrating the specification, modeling, and verification power of Orbits. We finish with a discussion of some techniques to reduce the size of the state space.

Chapter six introduces timing to our model and describes how timing affects our behavioral interpretation. We compare our timing syntax and semantics with other forms of timed Petri nets and explain how our operational semantics extended to timing preserves the trace structure theory of orbital nets. We show how using only discrete event occurrence times yields exactly the same results as using continuous time for certain properties, including trace structure conformance. Finally, we present some examples of delay buffers and circuit specifications using timing and show some novel techniques that Orbits introduces.

Chapter seven continues the discussion of timing by describing some new algorithms for analyzing orbital nets. These new algorithms are orders of magnitude more efficient for timing verification than existing methods. We introduce a method of normalization that keeps the state space finite without imposing special restrictions on the systems that can be examined. We present a sequence of techniques for state space exploration that dramatically reduce the number of edges in the timed reachability graph. We show that these techniques are general and apply to many different forms of timed automata. We describe techniques that can be used to answer some common questions that arise in timing analysis. Finally, we discuss how these results apply to simulation.

Chapter eight introduces data flow into our representation; we show how to model values and operations in a circuit description. We give examples of how verification can proceed in the presence of data flow. Chapter nine introduces the relationships between data flow and circuit structure and shows how Orbits can specify circuits that combine complex relationships between control, data flow, and timing.

Chapter ten summarizes our results, discusses how Orbits can be improved to be more efficient and more usable, and touches upon some of the many questions that remain to be explored.

1.6 Related Research

This research relates to prior work in three major categories: rich representations and specification languages, synthesis and verification of asynchronous designs, and real-time verification techniques.

1.6.1 Previous rich representations

The best examples of timing and structure annotations in a conventional synthesis tool are the work done by J. A. Nestor at Carnegie-Mellon on the System Architect's Workbench (SAW) [39] [49] and the work done by David Ku at Stanford on Hercules/Hebe [25] [26]. Nestor added timing constraints between ISPS statements in the input specification, and the scheduler considered only schedules that obeyed these timing constraints. Because this work was targeted to the SAW framework, it made no distinction between timing constraints that the circuit must obey and the timing constraints that the environment could be assumed to obey. The ISPS input language suffered from a premature coupling of timing and function; with the timing annotations, input and output statements, and functional behavioral description interspersed throughout the program, various data flow and other optimizations could not be made. Ku's work included the specification of timing constraints in a C-like language, but focused on rescheduling multiple-cycle operations to minimize control costs within a synchronous design framework; we construct much more general specifications, but do not consider scheduling or synthesis.

A number of hardware description languages have recently become popular in industry, including VHDL [14], Verilog, and UDL/I [17]. Each of these is essentially a simulation language that allows specification of timing delays in various ways. All of these languages allow only low-level behavioral timing delays to be specified, rather than timing constraints on input and output wires or environmental timing behavior. It is possible to say that a nand-gate has a delay of fifteen nanoseconds, for example, but not that the response time from the strobe to the acknowledge must be less than one hundred nanoseconds, let alone that the cycle time from loop to loop of this algorithm must be less than three microseconds. It is possible to check for setup and hold violations given a circuit already designed, but it is not possible to describe the setup and hold requirements that either the circuit or the environment must obey.

Borriello [7] developed a graphical specification methodology based on timing diagrams for describing and synthesizing transducers. His work was limited in the types of behavior that could be expressed, and he only considered synthesis.

Rich representations have also received recent attention. Such research is being done by Amon and Borriello [2] [3], Knapp and Winslett [23], Zebo Peng [42], and P. A. Subrahmanyam [47], who each have their own representation with its own tradeoffs.

Zebo Peng's representation is based on Petri nets, with statements (or sequences of statements) associated with places. In essence, he uses Petri nets to represent the control flow of a program, and Petri net places to represent basic blocks; traditional data flow, scheduling, and resource allocation algorithms are applied. No consideration is given to general timing constraints or structure.

Another representation for hardware is the temporally annotated labeled event structure (TALES) by P. A. Subrahmanyam. These graphs separate the underlying clocking scheme from the behavior of the circuit and thus allow the designer to easily experiment with different clocking schemes or even a mixture of several clocking schemes. It also promotes the use of abstract communication between processes that can later be instantiated by physical communication channels. The research is supported by a precise mathematical formalism. It only deals with events and control flow, however; data flow is not handled. In addition, the syntactic constraints on the initial specification are stifling; it is impossible to specify, in a timing-independent manner, such common paradigms as reading the next input while you are computing or generating the current output.

Gaetano Borriello and Tod Amon have introduced a unified behavioral and structural representation called Operation/Event Graphs. These graphs resemble Zebo Peng's representation, except that timing annotations can be added between events, and the model of tokens flowing through the net is abandoned. In order to deal with timing annotations from loop to loop, and also to help handle complex timing constraints in a synchronous system, the idea of "ancestor" events and a new calculus are introduced. This calculus provides precision in specifying timing constraints at the expense of a great deal of complexity; use of timed Petri nets and tokens solves the same problem in a much cleaner way. Even though a formal calculus is used for the timing constraints, the semantics of the representation is far from clear, and there are many possibilities for ambiguity. Finally, the way in which computation and events are interspersed prematurely restricts the design space.

David Knapp and Marianne Winslett from the University of Illinois have had significant success with the Design Data Structure (DDS), introduced back in 1985. This formal model of circuits and specifications addresses timing, behavior, and partial structure, but all in the context of a synchronous system at the register transfer level. The timing considerations are limited to standard cycle by cycle analysis common in synchronous synthesis, and no consideration is given to interface specification. Nonetheless, they present a formal system for the representation of both specifications and implementations. Based on this representation, they present algorithms to verify compliance of the implementation to the specification and to repair designs that are modified or broken by the designer. This is one of the most comprehensive representations and also one of the most formal, but it is of limited use because it only applies to designs that are already completed down to the register transfer level.

1.6.2 Synthesis and verification of asynchronous circuits

The second type of related research is in synthesis and verification of asynchronous circuits. Asynchronous circuit design is primarily concerned with circuits that work no matter how fast or slow individual components are; by building small circuits that function correctly under any delay assumptions and composing them in specific ways, they can completely ignore timing considerations when constructing larger and more complex circuits. Issues of clock distribution and skew, path lengths, and timing in general can be largely ignored.

Much of this work is based on Signal Transition Graphs (STG), a Petri net based formalism used to describe the state graph of an asynchronous system. Indeed, the control portion of orbital nets looks much like an STG, although orbital nets are both richer and more expressive. The main limitations of recent asynchronous research have been in the narrow domain considered; such research is concerned primarily with building timing-independent state machines from a graph model specification. We build on this work, extending the formalisms to handle more general circuits with real timing constraints, data flow, and structure.

Asynchronous circuit design styles can be separated into three major categories: delay-insensitive, speed-independent, and self-clocked. A delay-insensitive circuit operates correctly no matter what delays are given to any gate or any wire. While much early work on synthesis concentrated on delay-insensitive designs, it was soon realized that there were significant limits on the types of circuits that could be built using this timing assumption [29]. The weaker assumption of speed-independence, where the circuit is designed to be correct under any gate delays, but where wire delays must be negligible with respect to gate delays, has been the focus of some more recent synthesis work [33] [13] [27] [12]. Delay-insensitive and speed-independent circuits must be hazard-free, and this requirement can require additional circuit complexity. Self-clocked methods [41] are an intriguing combination of synchronous and asynchronous design; they generate their own clock signal based on internal state changes, and almost all hazards are eliminated by gating state changes through a register enabled by this clock. A common desire expressed in much of this research is for a simple but descriptive specification language. In particular, signal transition graphs become difficult to manipulate once a specification exceeds a certain size. In addition, all of these techniques apply only to control and do not consider real-time delays or data flow. Speed-independent and self-clocked circuits, as well as synchronous circuits, have timing assumptions that must be verified once the final design is laid out. Indeed, some of the synthesis algorithms can be extended to take into account real-time delays in order to generate simpler circuits [38]. A recent overview of many of these algorithms is given in [18].

We also draw on prior research in verification; this is especially important when trying to define a behavioral semantics for a representation intended to represent allowable (for specifications) or possible (for models) behavior. David Dill's dissertation [15] is the basis of our behavioral semantics. Indeed, Orbits subsumes his verification program `aver` as the special case of verification with no timing and no data flow.

1.6.3 Previous research in real-time systems

Burch extended the theory of trace structures to timing [11], but his verification system was not hierarchical, and it lacked a proof of the relationship between discrete and continuous time for his circuit models. Finally, his system was subject to extreme state explosion due to the separate consideration of each possible discrete delay at each point in the verification.

Prior research in geometric methods for timed state space exploration is also relevant to the timing portions of this dissertation. Berthomieu and Diaz [6] explore the state space of time Petri nets, but their algorithms do not take concurrency into account. Lewis [28] uses geometric methods on timed state diagrams, but each step takes $O(n^3)$ time; our step operation is $O(n^2)$. His algorithm considered an exponential number of successors to each state in his state diagrams; we only consider a linear number of successors. Van der Aalst [51] presents a fast and efficient geometric method for the analysis of Timed Colored Petri nets, but his technique is a conservative approximation that might return a false negative; our techniques are exact for safety properties. None of these techniques is directly applicable to circuits because of a lack of distinction between input and output events, and none of these considers the comparison of a specification and an implementation.

Two recent dissertations from Stanford by Henzinger [19] and Alur [1] rigorously consider real-time systems and verification, with special attention to computability and complexity. These works are definitive, citing many problems and pitfalls with existing formalisms and accurately identifying formal logics and automata for which verification is computable. But neither dissertation is concerned with building an efficient verification system, the property of receptiveness, or a conformance relation suitable for circuits.

Chapter 2

Orbital Nets

The effort really to see and really to represent is no idle business in face of the constant *force that makes for muddlement. —Henry James, 1843–1916*

This chapter introduces orbital nets. We discuss the operational aspects and properties of Petri nets and describe how Petri nets form the basis of orbital nets. We then present a number of examples, relating models that can be constructed with orbital nets to models used in previous research.

We must consider what we mean by behavior, and we must adopt a syntax capable of describing a set of behaviors. It turns out that Petri nets provide a natural representation of the control aspects of a design, so orbital nets are based on Petri nets. Others have used Petri nets to model circuits and specifications. The success of Orbits is due to its use and extension of Petri nets for synchronization, receptiveness, timing, and data flow while still supporting a formal behavioral semantics.

2.1 Events and Behavior

To formalize our concept of a behavior, we must define the events that comprise a particular behavior. For circuits, the most natural set of events is the set of possible value changes on wires. Circuits communicate through wires (or groups of wires that we will often consider collectively as a single wire), that carry both commands (strobes, interrupts) and data (the number five, the next thirty-two bits to display on the video screen). Values are indicated through the voltage potential on the wire—for instance, in some circuits, a voltage above 2.0 volts is considered the value one or true, and a voltage below 0.7 volts is considered zero or false.

We divide wires into two categories depending on the type of data they carry. A *signal wire* signals some change of state or the start of some operation; each high-to-low or low-to-high transition of the wire is important. A strobe or acknowledge is such a wire. A *value wire* is used to carry a specific value from one portion of the circuit to another, often as part of a data path. On value wires, when the value becomes available and when it is no longer available are the transitions of interest, rather than the specific value itself. This separation between signal and value wires reflects exactly the difference between control and data flow in a circuit.

This is an important distinction between types of wires for modeling. When a signal wire indicates a change of state or a command, there is a single voltage change on the wire. In general, every such voltage change is important in describing the behavior of a circuit. In addition, it is precisely this change of potential which communicates information from circuit element to circuit element.

With value wires, only the potential on the wire during a particular interval of time is important. Indeed, if two separate items of data, each with the same value, are communicated sequentially over a value wire, the voltage on the wire might not change; in general, changes on value wires themselves cannot indicate or control the flow of data. If the values differ, but the value wire is a bus or collection of multiple wires, the times at which the voltage on each individual wire changes are typically different—so a flurry of voltage changes occur, of which only the final result is important. The events on a value wire are the data becoming stable and valid and the data becoming invalid.

For the present time, we focus on signal wires, but we will return our attention to value wires in a little while.

If we define an event in a circuit to be a change of value on a wire, then we must define the state of the circuit as an assignment of values to all of the wires. States and events are the dual concepts of almost any form of system analysis—events are changes of state, and state defines which events are permitted or possible.

Circuits tend to be highly concurrent systems, where both state and events exhibit a high degree of locality. Indeed, one measure of the efficiency of a circuit is how many different parts of the circuit are working—or observing, reacting to, and causing events—at the same time. Locality is useful information, and we would prefer that our representation not hide it from us.

For instance, in a given state, we might have a large set of enabled events. Locality distinguishes events that are in conflict (only one or the other may occur, but not both) from events that are concurrent (both may occur in any order or at the same time). State-machine based formalisms normally hide such information because the state is a single global entity. Petri nets maintain such information because the state of a system is separated into small pieces. In addition, we have available a large body of research on Petri nets and a large set of tools and techniques for the analysis of Petri nets. For these reasons, we have chosen Petri nets as the basis of our representation.

We shall find that this is an excellent choice; the structure of Petri nets enhances our ability to specify, simulate, and formally verify circuits. We shall extend Petri nets in a number of ways, but for now we restrict ourselves to labeled Petri nets.

2.2 Petri Nets

Formally, a Petri net N is a four-tuple (P, T, PE, M_0). The net forms a bipartite directed graph with two types of nodes: T is the set of transitions, drawn in our diagrams as thick bars, and P is the set of places, drawn as circles. The set of edges is given by PE, drawn as a solid arrow; this set is of type $T \times P \cup P \times T$ [43]. An arrow with a head at both ends indicates two edges, one in each direction.

The state of a Petri net is represented by a marking M, which is an assignment of some nonnegative number of tokens, drawn with large black dots, to places. M_0 is the initial marking of the net.

The postset of a node is the set of nodes that are immediate successors to that node; the preset of a node is the set of nodes that are immediate predecessors. The postset of node n is written as $n\bullet$, and the preset of node n is written as $\bullet n$.

A transition t is enabled to fire in marking M if each input place to that transition has at least one token in it; we write this as the predicate $M[t\rangle$. A step μ, consisting of a non-empty set of transitions, is enabled to fire in a marking M if for each place in the Petri net, the number of tokens in the place for marking M is greater than or equal to the number of transitions in μ with that place as input. We write this as $M[\mu\rangle$.

A transition is fired by removing one token from each input place and adding one token to each output place, resulting in a new net marking M'; we write this as $M[t\rangle M'$. A step is fired by firing the component transitions in any order. We also write the enabledness of a sequence of steps or transitions as $M[\alpha\rangle$, and its resulting marking M' with $M[\alpha\rangle M'$ [53].

A firing sequence of a Petri net N is $\alpha \in (2^T)^*$ such that $M_0[\alpha\rangle$; the set of all such firing sequences is represented by Q. The reachable markings RM of the Petri net is the set of all M such that $\exists \alpha \in Q$ and $M_0[\alpha\rangle M$. It will turn out that, by the way our nets are constructed, no reachable marking will ever assign more than one token to a place. Such Petri nets are called *safe*, and the markings may be represented by a subset of places. We write the empty sequence as ϵ and the empty set as ϕ.

We define *enabled* as a function that takes a marking and returns a set of steps: $\mu \in enabled(M)$ if and only if $M[\mu\rangle$. We define *fire* as the marking returned by firing a transition, step, or sequence of transitions or steps: $fire(M, \alpha) = M' \equiv M[\alpha\rangle M'$.

In addition, we define a predicate *fail* over members of Q that indicate whether a firing sequence is a failure or not. Initially we will define failure as any sequence that results in a marking with no transitions enabled, or with some token in a place with no output edges: $fail(\alpha) \equiv (\alpha \in Q \wedge \forall t, \alpha.t \notin Q) \vee (\exists p.p \in fire(M_0, \alpha) \wedge p\cdot = \phi)$. One general characteristic of *fail* is that if a prefix of a firing sequence is a failure, so is the whole firing sequence.

If the firing sequence $\alpha\mu\beta$ is in Q, and μ has more than one element, then for every $t \in \mu$ the firing sequence $\alpha t(\mu - t)\beta$ is also in Q and results in the same final marking; thus, every firing sequence with steps containing more than one transition can be mapped to one or more other firing sequences consisting of single transitions. Every step defines a class of mutually concurrent transitions. Many existing Petri net algorithms take advantage of this concurrency information in order to calculate results more rapidly; we shall develop one such technique in chapter seven. On the other hand, this concurrency information is not essential; considering only interleavings of single transition firings yields the same set of reachable markings as does considering all possible step firing sequences. This is the *interleaving property* of Petri nets. It allows us to only consider enabled transitions rather than all enabled steps when calculating reachable states. Since the number of enabled transitions is linear in the size of the Petri net, and the number of enabled steps can be exponential in the size of the Petri net, this leads to substantial computational savings. This property is maintained as we extend orbital nets to handle timing and data flow. We define the operation $|_T$ that restricts a set of firing sequences to those that consist only of single transitions.

As all of the work based on signal transition graphs has shown, Petri nets themselves provide a useful representation for certain design problems. As with signal transition graphs, we label Petri net transitions with events, where an event is either a rising or a falling edge (a change from a low voltage to a high voltage and a change from a high voltage to a low voltage, respectively) on a particular wire. While signal transition graphs are usually restricted to a subclass of Petri nets known as marked graphs, where all of the places have an preset and an postset of size one, we shall not impose any such constraint. Figure 1 shows the simplest nets for a signal wire and a value wire.

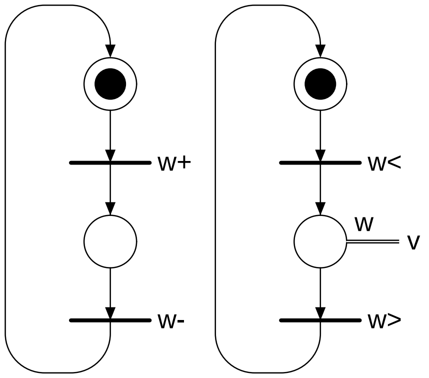


Figure 1. A signal wire and a value wire.

2.3 Orbital Nets

Orbital nets are extended, labeled Petri nets. Each transition is labeled with zero or more events; each transition or step firing models the occurrence of the corresponding events. We will discuss the behavioral semantics of orbital nets in chapter four.

Orbital nets are created from Petri nets by adding a number of additional nodes, edges, and functions. We will give an overview of the syntax in this section. We have already introduced the basic Petri net formalism:

P	Places
T	Transitions
PE	Petri edges
M_0	Initial marking

Labeling transitions requires the following extensions, described in chapter four:

I	Input actions
O	Output actions
Σ	$I \cup O$; all actions
lab	Labels; $T \rightarrow \Sigma^*$

Composition and receptiveness require two additional node types:

EP	Epsilon places
RP	Receptiveness places

Timing will consist of two additional functions:

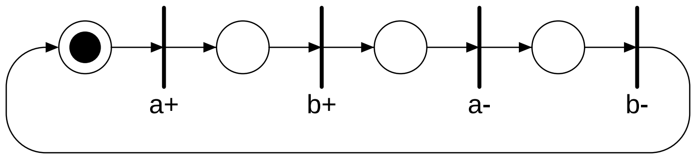


Figure 2. A buffer?

treq	Timing requirements; $P \rightarrow (duration, duration, timing\text{-}type)$
ignore-timing	Ignore timing requirement; $P \rightarrow Boolean$

Data flow requires three additional node types, an additional edge type, and one new function:

F	Data flow function nodes
DR	Registers: data flow assignment
J	Joins: data flow storage
DE	Data flow edges
func	Data flow functions; $F \rightarrow arithmetic\text{-}func$

The following behavioral aspects of an orbital net can be derived from the structure given in the previous tables:

Q	Firing sequences
S	Success set
F	Failure set
L	Language; $S \cup F$; all event sequences
TS	Trace structure
RM	Set of reachable markings

2.4 Inputs and Outputs

It is crucial to distinguish inputs and outputs when reasoning about circuits. Consider the net in figure 2. It describes the relationship between events on two wires, a and b. The net does not specify whether a or b are inputs or outputs to the circuit element in question. There are four possible interpretations of this net.

The first possibility is that both a and b are outputs. In this case, the net describes the behavior of a circuit element generating events on a and b, and it says that the four events shall repeat in the sequence $[a+, b+, a-, b-]$, and that the circuit shall not generate any behaviors except this one. This net may be used either as a low-level model of the behavior

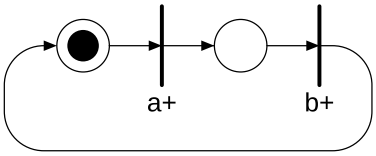


Figure 3. A 2-phase buffer or inverter.

of such a circuit element, or it may be used as a specification of how such a circuit element shall function.

When a is an input and b is an output, the net models a buffer with arbitrary delay. In its initial state, it waits for a rising transition on the input wire a. At some time after the circuit observes this event, the output wire b rises; this describes the behavior of the circuit. Between the time that a rises and b rises, no event is allowed to occur on a; this is a constraint on the environment. The same occurs for the falling edges, and then the behavior repeats. Later we formalize the idea of a constraint on the environment when we introduce receptiveness and failures.

This is only one possible model of a buffer; other models may allow more input behaviors and characterize the behavior of the buffer under adverse input conditions. Because we use a buffer as our primitive delay element when modeling circuits, we shall construct many different models for it.

When a is an output and b is an input, the net models an inverter with delay, with a pending output event in the initial state.

When both a and b are inputs, then the net represents only constraints on the environment, since the environment is the only agent generating events. When we introduce mirroring and receptiveness, we will be able to show how this specification can be used to check the behavior of the environment.

In the previous example, the events we modeled were rising and falling edges on a wire. For some design styles, it is often useful to model just an edge on a wire, without specifying whether it is rising or falling. Circuits designed in this way are sometimes faster; two-phase handshaking lacks the reset phase that can slow down four-phase handshaking. In our models, there is no specific interpretation given to the labels, so it is certainly possible to have $a+$ represent either a rising or falling edge, and to build libraries of models of circuit elements that use this convention. Figure 3 shows one such model; it can represent either an inverter or a buffer, since their behaviors are identical if the polarity of the events is not important.

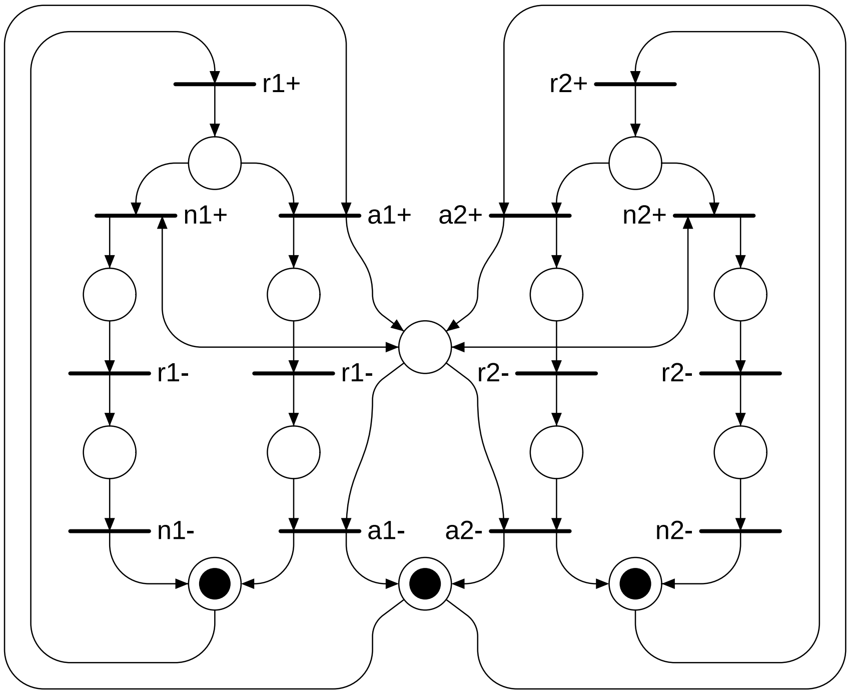


Figure 4. An arbiter with reject. Note how the interface specifications for each of the clients comprise the left and right sides of this figure, while the mutual exclusion requirement is embodied by the middle two places; the composite behavior is the intersection or composition of these three behaviors.

2.5 An Example Specification

Petri nets and, in particular, signal transition graphs, have been shown many times to be effective specifications of circuits. We present just one such example: an arbiter with reject. This is precisely the same Petri net presented by Nowick and Dill in [40]. We present this example to show that our representation works as well for specification as for modeling circuit elements.

An arbiter with reject has two clients, and it communicates with each of them with a request wire r, an acknowledge wire a, and a negative acknowledge wire n. Each client asserts r and waits for a positive edge on either a or n. If a positive edge on a is generated by the arbiter, that client has the resource and may use it. After returning the resource, the client lowers r to return use of the resource to the arbiter, and the arbiter will in turn lower a. If the resource is busy when the request is originally made, then a positive edge on n will be observed. The client then lowers r and waits for n to fall. The arbiter ensures that only one of the clients has the resource at a time. As we see in figure 4, this is easily specified or modeled in our representation.

state	action	next	state	action	next	state	action	next
1	$r1+$	2	10	$n2-$	11	20	$r1-$	21
1	$r2+$	16	11	$r2+$	4	20	$a2-$	26
2	$r2+$	3	11	$r1-$	12	21	$a2-$	22
2	$a1+$	11	12	$r2+$	5	21	$n1-$	25
3	$a1+$	4	12	$a1-$	1	22	$r2+$	23
3	$a2+$	18	13	$r1+$	14	22	$n1-$	1
4	$r1-$	5	13	$r2-$	8	23	$a2+$	24
4	$n2+$	15	14	$r2-$	9	23	$n1-$	16
5	$n2+$	6	14	$a1+$	15	24	$r2-$	21
5	$a1-$	16	15	$r1-$	6	24	$n1-$	17
6	$r2-$	7	15	$r2-$	2	25	$r1+$	19
6	$a1-$	13	16	$r1+$	3	25	$a2-$	1
7	$a1-$	8	16	$a2+$	17	26	$r2+$	27
7	$n2-$	12	17	$r1+$	18	26	$r1-$	22
8	$r1+$	9	17	$r2-$	25	27	$r1-$	23
8	$n2-$	1	18	$r2-$	19	27	$a2+$	28
9	$a1+$	10	18	$n1+$	28	28	$r1-$	24
9	$n2-$	2	19	$n1+$	20	28	$r2-$	20
10	$r1-$	7	19	$a2-$	2			

Figure 5. A state machine representation of the arbiter with reject.

There are two separate and largely independent interfaces, one for each client. With Petri nets, this is easily described and easily generalized for an arbitrary number of clients. A state-machine based representation would be more complex and would grow much more so as the number of clients increased. For example, the state-machine description of the above net is given in figure 5.

2.6 Modeling Primitive Gates

The Petri net formalism is a general model of concurrent computation, and thus it is possible to have many different Petri nets that represent the same or similar behavior. Often it is not immediately obvious what the differences between similar models are. We now present several different representations of a simple two-input and-gate and discuss some of the differences between them.

One of the primary questions we explore is whether a physical gate should be modeled with the delay as an integral part of the gate model, or as some sort of instantaneously fast gate with delay buffers attached to the inputs, outputs, or both inputs and outputs. In this section and in chapter five, we shall see that the latter is a better alternative. We will examine how the size of the model grows as the number of inputs grow and note whether each model is implicitly receptive. Informally, a model is receptive if it allows any input to

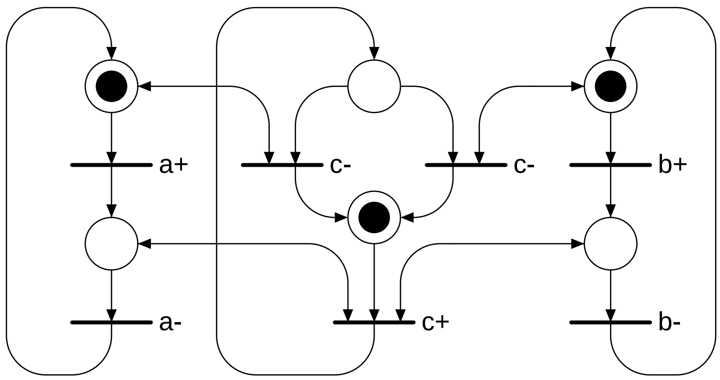


Figure 6. First model for the 2-input and-gate.

occur in any state, even though these inputs may lead to failure traces. Thus, our buffer examples were not receptive, as they did not allow another input while an output was pending.

All of the gate models will have the input wires a and b and the output wire c.

Our first model, given in figure 6, is based on a model for the and-gate presented by Shapiro [46]. This is an elegant model; it is receptive and easily extended for any number of inputs; the total number of places, transitions, and edges for a model with n inputs is $2n + 2$, $3n + 1$, and $8n + 2$, respectively. In this model, there is an arbitrary delay between an input and any output that it causes. If a hazard occurs on an input, this model nondeterministically chooses whether to ignore the hazard or to propagate it to the output. In any case, it will never flag the hazard as an error. Under certain conditions, this might be the semantics we want. Often, however, we want more deterministic behavior; if a hazard occurs, it is an error, or the circuit behaves in a particular way.

The second model, given in figure 7, is based on the model that David Dill used in his dissertation [15]. This model is somewhat larger than the previous one, requiring 2^{n+1} places, $(2n + 1)2^n - 2n$ transitions, and twice as many edges as transitions. This model is not receptive as presented here, but it can easily be made receptive by adding another $2n$ failure transitions; these transitions will fire only if there is a hazard.

In both of these models, the delay from input to output is an integral part of the net. But there are many different models for delay in a circuit. One such model, where all hazards are flagged as errors, is implicit in Dill's and-gate. Another, allowing hazards to either be ignored or propagated by the gate, but not making them explicit failures, is implicit in Shapiro's model. But we might need to specify that in the presence of a hazard, the gate can go into a particular type of failure mode where it can generate arbitrary transitions on the output. Or, we might need to specify a particular range of delay values from an input transition to the output transition it causes, and treat pulses shorter than a certain width

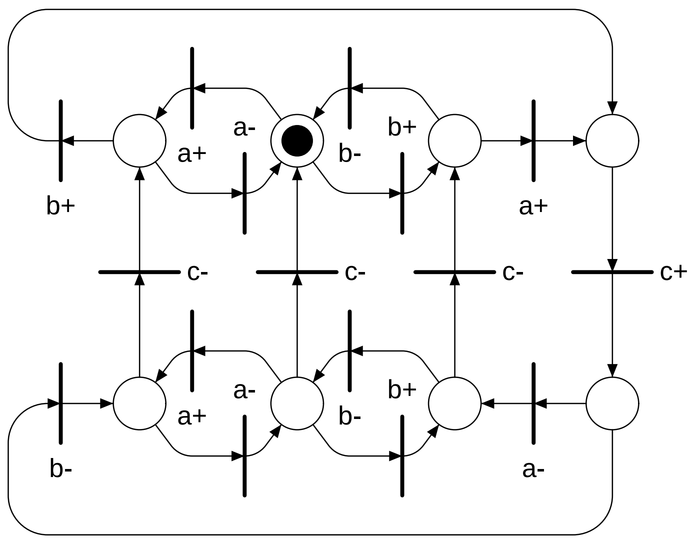


Figure 7. Second model for the 2-input and-gate.

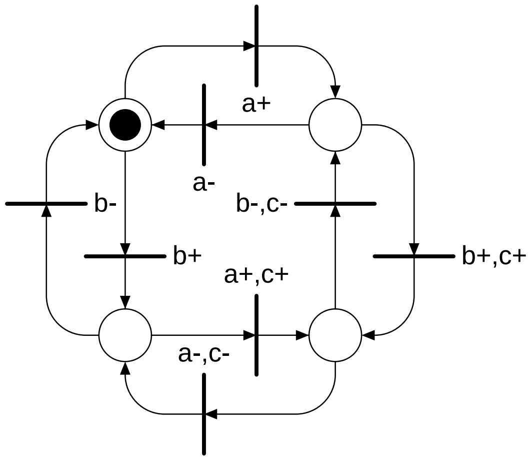


Figure 8. Third model for the 2-input and-gate.

differently. For each of these cases, it is not immediately clear how to modify the above models to get the desired behavior.

It is useful, therefore, to separate modeling of the logical function of the gate from the modeling of the delay aspects. Thus, we might use an instantaneous model of an and-gate, as shown in figure 8, and compose this with a delay element that provides the semantics we desire. Then, whenever we wish to model delay in a particular way, only the model of the delay element need be changed; all of the gates and other primitive elements remain the same.

This style of model requires 2^n places and $n2^n$ transitions for an n-input gate and is receptive if the net that it is composed with on the c wire is receptive. If we use a net such as the one in figure 2 as a buffer to model the output delay, then we end up with a net that has precisely the same behavior as Dill's model. If we compose it with a net such as that

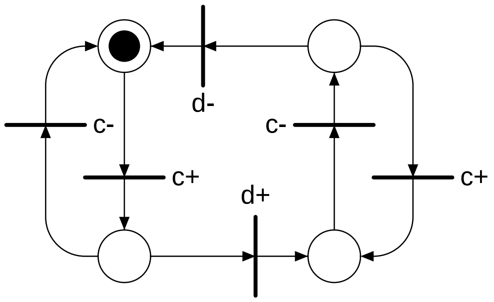


Figure 9. A delay buffer that nondeterministically either forwards or ignores hazards.

in figure 9, then we have a net with precisely the same behavior as that generated by our first model of the and-gate.

The examples we have presented are extremely simple, and the nets used in each case are small enough to construct and analyze by hand. In the following three chapters, we will show how to construct much more complex models and specifications using a textual language, how to combine models together into a circuit, and how to compare the behavior of circuits and specifications.

Chapter 3

Textual Description

High thoughts must have high language.
—Aristophanes, c. 450–385 B.C.

A representation is only useful if the designer can construct and write detailed specifications and models. This chapter presents our language for creating specifications and circuit structures in our representation. We will discuss the basics of our language, its translation to nets, and how to create finite state machines.

3.1 Textual Languages versus Graph Representations

Graph models have long been used as an initial specification of a circuit. Two common representations are simple state machines, either synchronous or asynchronous, and signal transition graphs, widely used in the asynchronous design community. But a pure graphical representation is only adequate for simple designs; once the graph grows above twenty or so nodes, it becomes too complex to work with directly. While hierarchy and partitioning, two graph decomposition methods, can be used to partially alleviate this problem, graphical descriptions remain unwieldy for the designer to create, interpret, and modify.

Designers are often more comfortable working with a textual language for circuit descriptions. Indeed, the use of a textual language allows the results of years of programming language design to be applied to the circuit description problem. Such well-understood concepts as parameterization, abstraction, identifier scoping, and clear syntax can be major benefits when building a circuit description.

Tools also benefit from the use of a textual language. Graphs constructed syntactically from a textual description are often better structured than graphs designed by hand in an ad-hoc manner, much as data flow and basic block analysis of Pascal programs tend to be much easier than the equivalent analysis performed on assembly language programs. For the specific case of generating a model based on Petri nets, such graph-theoretic properties as safety can be guaranteed by construction.

Using a text-based language as input allows the use of parameterization, macro facilities, partial evaluation, and procedure-based construction of the Petri nets. We present several examples of these ideas in this chapter.

There are many possible foundations for a language intended to describe graphs. One simple textual representation would be a list of the various nodes and the arcs connecting them. While simple and direct, this approach ignores many important advantages of high level languages. We prefer to leverage off the designer's existing understanding of programming languages.

Another alternative is to base the language on regular expressions. We shall see later in this chapter that this does not work as well as we desire; correct Petri net constructions for regular expression operations introduce unlabeled transitions that change the semantics of the net with respect to nondeterminism and receptiveness. Another possible foundation for the language is finite state automata, but the resulting descriptions are often far more verbose and unwieldy than is necessary. The language we present here draws upon different aspects of all of these alternatives. It is just one of many possible languages that might generate orbital nets.

Our language uses the simple S-expression model of Lisp; a description is nothing more than a (possibly nested) list. Such expressions are easy to parse and easy to understand. We do make a few concessions to syntax through the use of macros.

The language is only given meaning through the orbital nets that result. Each model or specification in the language is translated into an orbital net, at which point the textual description is no longer used. The orbital net contains all of the intended design information.

The translation from textual language to orbital net is done through an evaluation process. Evaluation of expressions in our language have the side effect of adding nodes, edges, and annotations to an implicit global net. The evaluation procedure takes as arguments the expression to evaluate, an *environment* in which to evaluate it, and a *current place* that serves as the starting point on which to build the graph; this place serves as the *initial place* of that expression. The results of the evaluation procedure are a possible value, such as a number, or a wire, or a circuit, and a possible *final place* that is the place that will be marked when simulation of the expression has terminated. Some expressions will return neither a value nor a final place (such as an infinite loop); others will only return a final place, and still others will return both a place and a value. The initial place and the final place might be the same place.

`let`	Extend a local environment
`def`	Extend the global environment
`annotate`	Create and labels a transition
`begin`	Start a sequence
`par`	Execute in parallel
`choice`	Make a choice
`forever`	Execute repeatedly
`if`	Execute conditionally
`when`	Execute conditionally
`guard`	Conditionally make a choice
`case`	Branch based on value
`sm`	Construct a state machine
`sm-ckt`	Construct a gate
`combinational`	Build a nested circuit
`make-wire`	Allocate a wire
`with-wires`	Allocate many wires
`wire-init-set!`	Specify the initial value
`wire-init-known?`	Determine if initial value known
`wire-init`	Obtain the initial value
`complete`	Declare synchronization groups
`process`	Complete infinite loop
`receptive`	Build a receptive circuit
`mirror`	Take the mirror of a circuit
`normalize`	Normalize a specification
`verify`	Calculate failure-freedom
`si`	Build a speed-independent circuit
`di`	Build a delay-insensitive circuit
`simulate`	Simulate an orbital net
`step`	Perform simulation steps
`delay`	Add timing annotations
`ignore-timing`	Indicate timing should be igored here
`show-constraint-place-results`	Display timing results
`make-reg`	Allocate a shared register
`set!`	Set the value of a register
`atomic-read`	Read the value of a register
`make-key`	Allocate a key
`!`	Send a value to a key channel
`?`	Obtain a value from a key channel
`value`	Observe the value of a wire
`setvalue`	Set the value of a wire
`!-value`	Send an observed value
`?-setvalue`	Receive a value and set a wire
`+`, `-`, `*`, `/`, ...	Arithmetic and Boolean operations

Figure 10. The language keywords.

The language includes all of the commands listed in figure 10. We will describe each in depth as we require it.

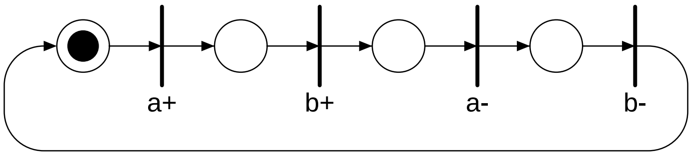


Figure 11. A buffer with input a and output b.

Let us consider again the simple net given in figure 11. This is perhaps one of the simplest possible nets. Since simple things should be simple, we can easily construct it with:

```
(with-wires (a b)
   (forever a+ b+ a- b-))
```

There are several different concepts that are needed to build this net. The first is the concept of an action, or event. From the perspective of the designer, these events occur on wires, so we also need the concept of a wire. We also need some sort of control flow, indicating the order in which the events occur and that the net forms a loop, repeating the event sequence over and over. We address wires and events, control flow, and loops in turn.

3.2 Actions and Wires

In our representation, actions are just labels on transitions—they do not have any particular meaning associated with them in regard to the execution of the net. Thus, it might be tempting to allow arbitrary textual labels on transitions in our language. In the circuit domain, though, those actions are value changes on wires—and wires are of primary importance, with a fair bit of information associated with them. One such item of data is the initial value of the wire; another is whether that wire is an input or an output to the particular circuit element under consideration. There are some structural requirements on circuits that depend on this information, such as that no two outputs are connected together, and that separate models referring to the same wire use the same initial value for that wire. Therefore, we provide a structured way of labeling transitions.

There is another reason to insist on a structured method of annotating transitions. Many errors that occur during the development of a model are typographical errors, or scoping errors, or other errors that cannot be detected easily with arbitrary textual annotations. Insisting that annotations make sense—that they refer to allocated objects with reasonable operations on these objects—helps catch modeling errors.

Therefore, we require that wires, the primary object on which actions will occur, be declared or allocated. The primitive we use to allocate wires is `make-wire`.

`(make-wire` *[init-value [name [suffix]]]*`)` Allocates a new wire and returns it. The *init-value* parameter is an arbitrary value—the value `()` may be used if the initial wire value is not known at allocation time. The *name* parameter is used to give this wire a symbolic name; if the parameter is omitted, `W` is used. If the name ends in a number, then a colon is suffixed as well. The *suffix* parameter is used to generate distinct wire names. If it is not given, then a unique integer is suffixed to the *name* parameter (these integers are allocated in increasing order). If it is given as `none`, no suffix is used. Otherwise, it should be an integer that is the suffix. After constructing the name of the wire in this fashion, the global environment is checked to see if a wire with that name already exists. If it does, then that wire is returned; otherwise, a new wire is allocated.

To use a wire in an expression, some identifier must be bound to that wire. The `let` primitive binds an arbitrary identifier to an arbitrary value, just like in Lisp:

`(let ((`*id0 exp0*`) (`*id1 exp1*`)` ... `(`*idn expn*`)) .` *body*`)` Evaluates all of *exp0* ... *expn* in parallel, and then binds the identifiers *id0* to *exp0*, *id1* to *exp1*, etc., during the evaluation of *body*.

For convenience, the following macro is also defined:

`(with-wires` *[*`global`*]* `(`*wire-spec* ...`) .` *body*`)` Allocates a set of wires. If `global` is specified, then the wires are allocated in the global environment so they need not be passed as parameters to subcircuits. Each *wire-spec* is either a symbol (such as `a`) or a list consisting of a symbol and an initial value (such as `(a 0)`). In either case, the symbol is used both as the identifier to bind the wire to during evaluation of *body* and the root symbolic name to give the wire. All wires allocated with one `with-wires` invocation will have the same unique numerical suffix.

Thus, the example net can also be specified as

```
(let ((a (make-wire 0 a)) (b (make-wire 0 b)))
   (forever a+ b+ a- b-))
```

Later in the chapter we discuss how the initial values of the wires can be set and used.

Now that we have wires, we need to be able to build transitions and label them with events on wires. The `annotate` primitive provides this functionality; it is the primary primitive used to allocate a new net transition.

(`annotate` *[event*]*) Allocates a new transition, annotates it with the given events, and adds it to the net we are constructing. If no events are given, a new transition is still allocated. The events must be well formed. No value is returned.

A well-formed event is a symbol created by concatenating an identifier with an *action operation character*. The identifier must be bound in the current scope to an object that permits the provided operation; wires are the only such objects we have defined so far. The action operation characters we permit on wires are `+` (usually to indicate a rising edge on the wire) and `-` (usually to indicate a falling edge on the wire). Other objects and action operation characters will be defined later.

As a handy shorthand, if a well-formed event is evaluated outside of an `annotate` command, it is treated as an abbreviation for an `annotate` operation on that event; this allows us to use the code as written above rather than the more cumbersome code shown below:

```
(with-wires (a b)
   (forever (annotate a+) (annotate b+) (annotate a-) (annotate b-)))
```

3.3 Sequences

The `annotate` primitive adds a new transition to the orbital net; we also must be able to allocate and add places. Places are added implicitly by the way events are related, whether in sequence, in parallel, or through choice. The translation is simple, and understanding how it works is the key to understanding the language.

At any point during the syntax-directed creation of a graph model from the textual language, there is a *current place*. A token existing in this place models the current step of the textual description to be the beginning of the expression we are about to interpret. When we start the translation, we allocate an initial place; as translation proceeds, Petri net transitions and places and other elements are added to the graph.

When an expression is evaluated, both the value result (if any) and the Petri net place (if any) corresponding to the state after the evaluation of that expression are returned. The Petri net place is an implicit return value; it is not directly accessible during the evaluation process. Any value result is directly accessible (for instance, it can be bound to an identifier by `let`).

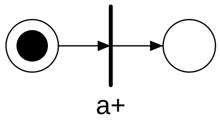


Figure 12. The net resulting from evaluation of `(with-wires (a) a+)`.

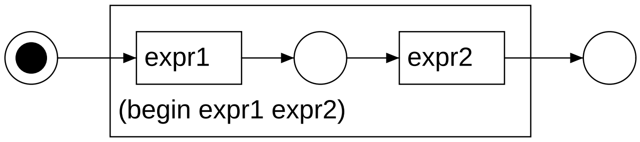


Figure 13. The construction for `begin`.

Thus, when `annotate` adds a transition to the net, it also adds a place, corresponding to the state of the system after that transition has fired. This new place then becomes the current place. Thus, evaluating the expression

```
(with-wires (a) a+)
```

yields the net shown in figure 12. The initial place is the place that is marked. The final place is the other place.

We can string together transitions and places with a sequence of expression evaluations; `begin` gives us this functionality.

(`begin` *exp0 exp1* ... *expn*) Evaluates a set of expressions in sequence and returns the value of the last expression.

The evaluation of a `begin` expression is straightforward. The first subexpression is evaluated, possibly resulting in additional nodes on our orbital net. Then, the final place resulting from the evaluation of that subexpression is made the current place for the evaluation of the remainder of the `begin`, which proceeds recursively in the same manner. The recursion is terminated when there are no more subexpressions, at which point the final place of the final subexpression is returned as the overall final place.

This construction is equivalent to creating an orbital net for each subexpression separately, and then merging the final place of each subexpression with the initial place of the subsequent subexpression. If we consider the subnet resulting from each expression evaluation as a black box that exposes only its initial and final places, then the result of evaluating a `begin` expression is shown by figure 13.

Many evaluations do not affect the net. For instance, the evaluation of an identifier bound in the current environment returns that value, but no changes are made to the net. For such simple expressions, the initial place and the final place are the same. Therefore,

```
(with-wires (a b)
   (forever a+ a a b+ b a a- b b-))
```

returns the same result and the same net as the original expression at the beginning of this chapter; the identifiers are evaluated, found to be wires, but their value is thrown away because the sequence operation only returns the last value. We will show the relationship between `begin` and `forever` a little later.

Expressions can be statically divided into three categories: those that generate a net that does not have a final place, such as an infinitely executing process; those that have a final place but do not return a value, such as an expression executed only for its side effects; and those that return one or more values, such as what is generally considered an expression. After the interpretation of an expression that does not terminate, there is no current place. After execution of either of the other two types of expressions, there is a new current place, although it may be the same as the current place we had before interpretation of the expression.

A simple expression is an expression that does not change the current place, does not add any additional transitions or places to the net, does not annotate the current place, and does not associate the current place with any identifier. We will discuss place annotations later. We distinguish simple expressions because they are evaluated at compile time and can be used to direct the construction of the net, rather than being events or control flow that are translated directly into the final net. This distinction recurs throughout the language.

3.4 Parallel Execution

In addition to specifying that certain expressions occur in sequence, we also need to specify that certain expressions occur in parallel. The `par` verb provides this functionality:

(`par` *exp0 exp1* ... *expn*) Evaluates a list of expressions and returns a list of the values of the expressions. This list is not a first-class object and cannot be directly accessed; only the side effects on the global orbital net are useful. This evaluation proceeds as follows: First, a new place is created for each of the expressions *exp0* through *expn*. Then, each of the expressions is evaluated after setting the current place to the corresponding initial place, and the final place (if any) of each subexpression is stored. After all of the subexpressions have been evaluated, some checks are performed. Either all of the subexpressions must have a final place, or none of them must. If each of the subexpressions is a simple expression, then all of the allocated initial places for the subexpressions are deleted, and the initial place of the overall `par` statement is restored as the final place. Thus, a set of simple

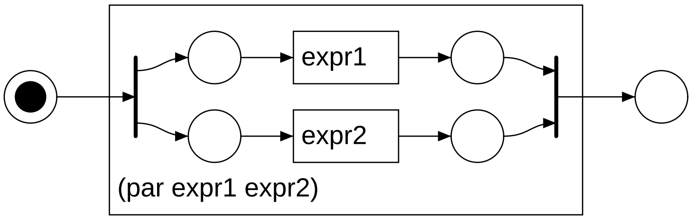


Figure 14. The construction for `par` when all expressions terminate.

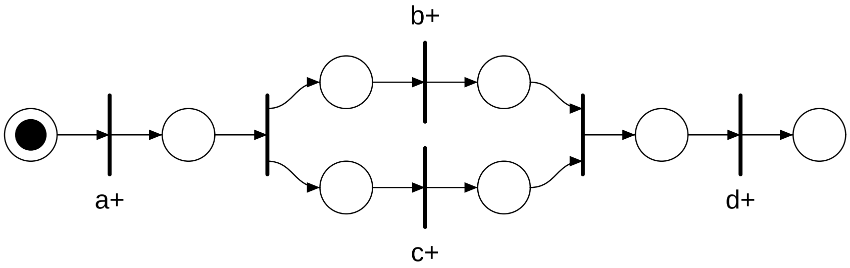


Figure 15. The translation of `(begin a+ (par b+ c+) d+)`.

expressions evaluated in parallel is itself a simple expression. If one or more of the subexpressions is not simple, then a new transition is allocated. This new transition has as its preset the initial place of the overall `par` statement and as its postset the initial places of the subexpressions. If none of the subexpressions has a final place, then neither does the overall `par`, and there is no final place. Otherwise, a second transition is allocated, with a preset of the final places of the subexpressions, and a postset of a newly allocated place. This new place is the final place of the overall expression.

The construction for `par` when one or more of the subexpressions is not a simple expression and all expressions terminate is shown in figure 14. The expression `(begin a+ (par b+ c+) d+)` is translated into the net given in figure 15.

This may not be what we want because of the two unlabeled transitions that are introduced. Unlabeled transitions are unobservable actions; there is no way for an external observer, who only sees actions and not unlabeled transition firings, to determine whether the state has changed or not. Consider the use of the above expression as a specification; it is probably incorrect for there to be some interval after $a+$ occurs during which neither $b+$ nor $c+$ is allowed to occur. These unlabeled transitions are almost always undesirable; yet, the straightforward Petri net construction for parallelism naturally introduces these transitions.

The `par` operation is also used to compose different portions of our model by composing a set of subexpressions that do not terminate. The construction for `par` under these conditions is given by figure 16. For instance, `(par (forever a+ a-) (forever b+ b-))` constructs a system with two pulse generators running in parallel. Our straightforward

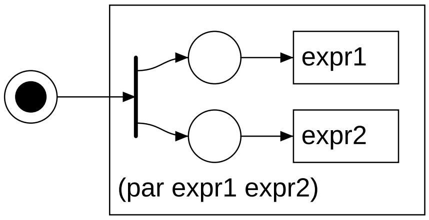


Figure 16. The construction for `par` when the subexpressions do not terminate.

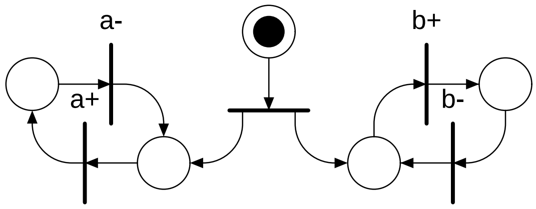


Figure 17. Two parallel processes composed with `par`.

construction, given this net, will generate the net shown in figure 17. We will discuss how communication and synchronization between separate finite state machines are handled in the next chapter.

In this case, we have an unnecessary unlabeled transition that will only fire once. We really do not intend to have this transition; it is only an artifact of using `par` to compose processes. Thus, after the entire expression has been evaluated and the entire net constructed, we perform initial marking propagation to eliminate all such transitions. Specifically, we search the net for a marked place with no input edges, no labels, and only a single output edge connected to an unlabeled transition with no other input edges. If we find such a place, we eliminate it and its output transition, and mark all places in the postset of that transition. We then iterate until no such marked places are left in the net.

One potential solution to the unlabeled transitions introduced by `par` is to split the current place into as many places as necessary, making copies of all the current input and output edges of the place. This can get messy, though, especially when we consider non-trivial control flow and choice. We do not, therefore, use this technique. There are similar problems in other areas of the translation that require similar solutions. A potential future research area would be to consider this technique when translating a textual specification into orbital nets.

3.5 Choice

Two events occurring in parallel is very similar to two events occurring in either of the two possible orders. We express this with `choice`.

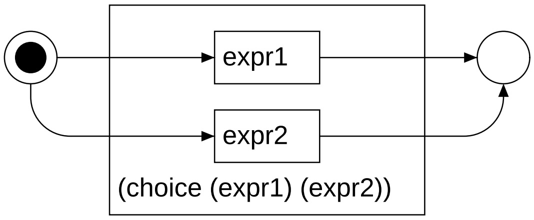


Figure 18. The construction for `choice`.

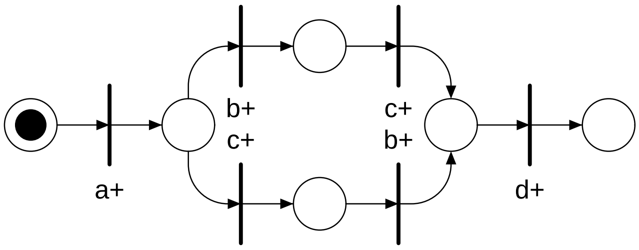


Figure 19. The translation of `(begin a+ (choice (b+ c+) (c+ b+)) d+)`.

`(choice (`*expseq0*`) (`*expseq1*`) ... (`*expseqn*`))` Models the selection of one of a number of different alternative behaviors. Each of the expression sequences is evaluated in turn (as though they started with `begin`) with the same initial place as that of the overall `choice` function. Then, a new place is allocated, and each of the final places of each subexpression is merged to that new place. This merging is not associative, and it takes place as follows: If the place being merged is a *virgin place* (meaning that it has no out edges, no labels, is not bound to any identifier, and it is not marked), then we delete that place and add all of its input edges to the place to which it is being merged. If the place being merged is not a virgin place, then we add an unannotated transition and a new place, which we then merge as before. The place that is the object of all of these merges becomes the final place.

The construction for `choice` is illustrated in figure 18. To eliminate unlabeled transitions in the example above we might write `(begin a+ (choice (b+ c+) (c+ b+)) d+)` instead of using `par`, yielding the net in figure 19. This is not an unreasonable alternative. The number of states is the same as for the pure `par` solution without the additional unlabeled transitions. Unfortunately, in general, use of `choice` to construct the possible interleavings often creates a much larger net. It also eliminates the concurrency between the two transitions; this concurrency can be exploited by many analysis techniques. Finally, such a transformation does not yield the desired results when we substitute more complex nets for the simple $b+$ and $c+$ actions.

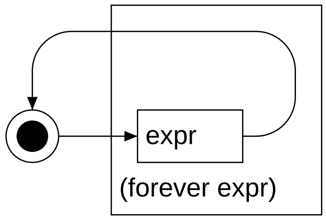


Figure 20. The construction for `forever`.

3.6 Loops

We have discussed everything in our example but `forever`. This command is a macro defined in terms of labeled `let`, which we describe next.

(`let` *label* `()` . *body*) Associates the identifier *label* with the current place during the evaluation of *body*. This place is treated as an operator that takes no arguments and acts as a go to statement. This makes it possible to build looping constructs. Unlike in Lisp, the operator bound to *label* cannot be used for general recursion, but only for tail recursion. We will extend this operator to allow new bindings, as in the unlabeled `let`, in chapter 8.

Thus, labeled `let` allows us to associate an identifier with any given orbital net place. The transfer of control is implemented with our place merge function described earlier; the current place is merged with the place associated with the identifier. The most common use, as described above, is with no additional identifiers or values bound, leading to a simple loop. The macro `forever` is used to create such loops:

(`forever` . *body*) Indicates that *body* repeats indefinitely. No value is returned, and the expression has no final place. Implemented as a macro that expands to `(let loop () (begin .` *body*`) (loop))`.

The construction for `forever` is given in figure 20.

3.7 Unlabeled Transitions and Regular Expressions

We have introduced sequencing, choice, parallel execution, and looping constructions. Intuitively, these would seem similar to the concatenation, alternative, interleaving, and Kleene star operations on regular sets. This intuition is strengthened because we are constructing finite state machines, which are equivalent in expressiveness to regular expressions. Unfortunately, the relationship between our net constructions and the regular expression operations is not as close as we might want. The primary difficulty is avoiding unlabeled

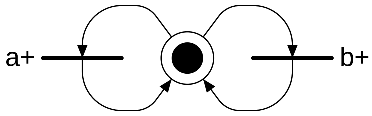


Figure 21. The translation of the expression `(choice ((forever a+)) ((forever b+)))`.

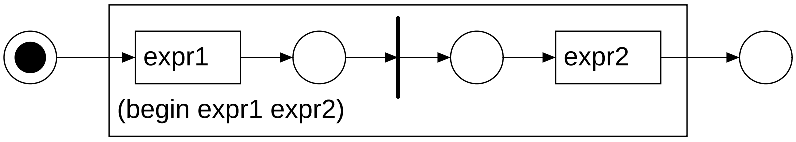


Figure 22. An alternate construction for `begin` that preserves the language generated by the net.

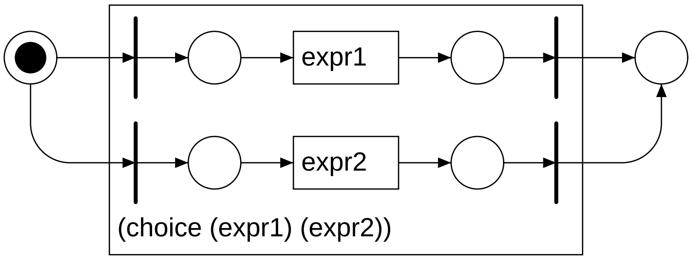


Figure 23. An alternate construction for `choice` that preserves the language generated by the net.

transitions; these unlabeled transitions change the semantics of our specifications in undesirable ways. This section discusses the differences between regular expressions and our orbital net constructions and explains why the selected constructions have been adopted.

Straightforward combinations of the `choice` and `forever` constructions as we have defined them might yield nets with different semantics than expected. On first glance, the expression `(choice ((forever a+)) ((forever b+)))` might be thought to yield a finite state machine that accepts the language $(a+^* \mid b+^*)$—but it does not. The net constructed according to the rules presented so far is given in figure 21; the language accepted by this net is $(a+ \mid b+)^*$.

The language is not the same because no explicit transition is introduced to indicate when a choice is made, or when control has passed from one expression of a sequence to the next expression. With the alternate constructions for `begin` or `choice` as illustrated in figure 22 and 23, respectively, the language generated by the Petri net would be what we expect.

If we use these constructions, the language is the same, but a large number of unlabeled transitions are introduced, resulting in additional states that yield false input errors.

Consider the sequence `(begin a+ b+)` where a is an output and b is an input. Our intention is to specify that at any time after a goes high, b can rise. The net resulting from

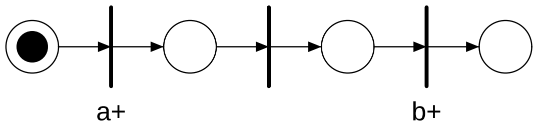


Figure 24. A bad sequence.

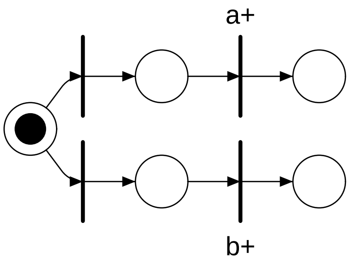


Figure 25. A bad choice.

our original `begin` construction generates the obvious and appropriate orbital net. If we use the revised construction above that preserves language equivalence, we would get the net pictured in figure 24. After the transition labeled $a+$ fires and before the unlabeled transition fires, the token is in a place from which there is no transition labeled $b+$, so the action $b+$ is not enabled. This is not what we intend.

Choice has a similar problem. If we mean that in a particular state either input a or input b can rise, the language-preserving `choice` construction would yield the net shown in figure 25. We intend that the environment chooses which input occurs first—either possibility is allowed. Yet, this net indicates that the choice is made nondeterministically; unlabeled transitions have no observable action associated with them, so they are completely nondeterministic. If the environment tries to make a choice different from the one made in this orbital net, then an input failure will result.

These unlabeled transitions can be eliminated with the standard operation on nondeterministic finite state machines that eliminates the nondeterminism and creates deterministic finite state machines. For the examples we have presented, performing this transformation would yield precisely the net desired. Indeed, as we shall see in chapter five, this facility is available with the `normalize` primitive of Orbits. But removal of nondeterminism can generate a potentially exponential number of states. In addition, each state in the resulting finite state machine might correspond to several states in the original finite state machine, and each state in the original might correspond to several states in the final finite state machine. When we introduce timing and data flow into orbital nets, this information will be associated with places; the remapping of states caused by the removal of nondeterminism will make it difficult to express timing and data flow correctly.

Finally, nondeterminism might be an essential part of our specification for certain circuits or systems. Explicitly eliminating all nondeterminism might destroy important information and yield false results.

So instead, we assume that the user of the language is aware of such effects from the constructions we use. For instance, the original example is easily given in a nondeterministic manner with `(choice (a+ (forever a+)) (b+ (forever b+)))`. We chose the constructions by considering the most common uses of these operators over a large set of examples and choosing the option that seems most reasonable. We also provide facilities for suppressing and creating transitions so that the user can build the net desired.

- The `par` primitive introduces an unlabeled transition at the beginning and end of the concurrent execution of the subexpressions if any of the subexpressions are not simple expressions. These transitions can usually be suppressed with the `::` operator, if desired.
- The `choice` and `guard` primitives do not introduce unlabeled transitions that would cleanly separate each possible continuation from each other. The first action of these alternatives is normally distinct, and each action usually terminates or branches back to some place before the choice has been made, rather than looping indefinitely after the choice.
- The `begin` sequencing operation does not put an unlabeled transition between events, since by far the most common use is for a simple sequence of actions.

The operator `::` can usually be used to eliminate undesired, unlabeled transitions introduced by `par`. This operator labels the current place as one that must be deleted by a post-processing phase over the final net, with the input transition and output transition of this place merged into a single transition. When the place is deleted, it must have only a single input transition and a single output transition. When the two transitions are merged, their actions are also combined. For example, the expression `(begin a+ :: (par b+ c+) :: d+)` generates the net shown in figure 26. If the place has multiple input or output transitions, then Orbits will signal the inappropriate use of `::`.

This operator can also provide a shorthand when using multiple annotations on a transition; `(begin a+ :: b+)` is exactly the same as `(annotate a+ b+)`.

In some situations, `::` does not suffice to eliminate unlabeled transitions. For instance, the expression `(begin (par a+ b+) :: (par c+ d+))` indicates that two unlabeled transitions, one introduced by each `par` statement, should be merged; the result is a single

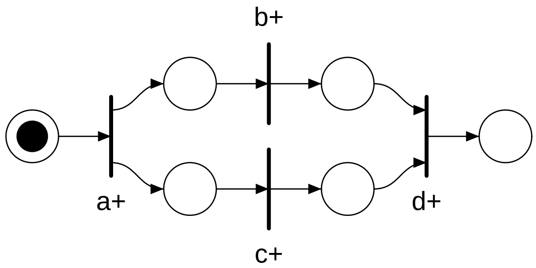


Figure 26. The translation of `(begin a+ :: (par b+ c+) :: d+)`.

unlabeled transition. In this and similar cases, interleaving or composition should be used to obtain the required net.

In general, if unlabeled transitions are desired, `(annotate)` will provide one anywhere; if they are not, they can often be suppressed with `::`. The user is responsible for this choice because the user is also responsible for understanding the existence, absence, or level of nondeterminism in models and specifications.

3.8 Evaluation and Environments

So far, the values we have allowed to be returned by the evaluation of an expression during the net construction have included wires and places. We have also seen an occasional numerical constant. We have not discussed how the arguments to an operator are evaluated. We have also not addressed how names can be associated with values or how wire initial values can be calculated and propagated throughout a circuit. We will now address these issues.

Numerical constants are a part of our language; when evaluated, they return their value. They are simple expressions since they do not affect the net we are constructing. In addition, we also allow some basic arithmetic operations on numbers, including `+`, `-`, `/`, `*`, `<<`, `>>`, `and`, `or`, and `not`. We also allow the comparison operations `=`, `<`, `>`, and `<>`; these operations return the value `1` if the comparison is true and `0` if it is false.

Evaluation proceeds as follows: If the expression being evaluated is a number, we return its value. If the expression is a symbol, we look up the value of that symbol in the current environment, and return the associated value. In both of these cases, the expression is a simple expression, and we do not modify the net.

If the expression is a symbol with no value in the current environment, we examine the last character of the symbol. If it is a legal action operation character and we are not currently evaluating an `annotate` expression, then we change the symbol to a list consisting of `annotate` followed by the symbol and evaluate that. If we are evaluating the arguments to an `annotate` operation, we strip the action operation character from the symbol and look for a value for the symbol in the current environment. If we do not find such a value, we indicate that an error has occurred. If we do find a value, we check that the value is an object of the correct type for the action operation character.

We also check for symbol ending in the character : and treat it as though the symbol were enclosed in a `delay` operation.

If the expression being evaluated is not a number and not a symbol, then it must be a list. If it is not a list, we indicate an error. Lists are evaluated as follows: If the list has zero length, then no value is returned; this is a do-nothing or no-operation that is often useful. Otherwise, the first element of the list is evaluated; it must be a simple expression (evaluation of the first element must not affect the net being constructed). The only types we allow for the first element are a primitive operation, a macro, a special form, or a place. If the first element is a macro, we perform macro expansion on the list, and then evaluate this result. If the first operation is a special form such as `if` or `let`, we handle it separately, possibly evaluating different portions of the remaining list depending on the special form.

If the first element is a primitive operation, we evaluate the remaining elements of the list as though the entire list started with `par`. If one or more of the arguments does not return a value, or does not terminate, we flag an error. If all arguments are simple expressions, we evaluate and return the result of that primitive operation over the result of the argument list. Thus, the application of a primitive operation to a set of simple expressions is itself a simple expression. For example, `(* 3 (+ 1 2))` is a simple expression. Special forms also are simple expressions if they can logically be considered such; thus, `(let ((a 3) (b (* 5 7)) (+ a b))` is a simple expression.

If the first element of the list is a place, then we evaluate the arguments as though the first element were a primitive operation. Each argument must be a simple expression and yield a known value. We then merge the current place to that place. Later, we will introduce the concept of operations on values that are not known at compile time.

Environments store the bindings between identifiers and values for different contexts. The environment is an implicit argument to all evaluations, and certain special forms can extend the environment. Different uses of environments give different scoping rules; we use environments in a way that provides lexical scoping.

There is a single global environment, which contains all of the predeclared operators. This global environment can be extended with the `def` special form:

`(def` *id val*`)` Associates a value with an identifier within the global environment. The *val* expression is evaluated, and a binding between the identifier *id* and that value is added to the global environment.

The *current environment* changes as functions are called and certain special forms are encountered. Specifically, the `let` construct extends the current environment with an additional set of bindings; these bindings will shadow other bindings in the previous environment if some of the newly bound identifiers already exist in that environment.

We overload `def` to allow us to define macros. We require that any use of these macros be fully resolved at net construction time. Nonetheless, macros have the scoping properties typically associated with functions.

`(def (`*id* . *formals*`)` . *body*`)` Associates a macro with an identifier in the global environment. The *formals* list is a simple list of zero or more symbols (as in `(def (foo a b) ...)`). When the macro is used as an operator, after evaluation of the arguments, the values of the arguments are bound to the identifiers in the *formals* list, and then the *body* is evaluated (with an implicit `begin`).

An additional form of `def` allows us to declare objects that are *circuits*, with a specific set of input and output wires:

`(def (`*id* `(`*inputs*`) (`*outputs*`) .` *extra*`) .` *body*`)` Associates a circuit with an identifier in the global environment. The identifiers in *inputs* and *outputs* can only be bound to action objects, and during evaluation of the body of the macro, objects from the first list are treated as input objects, and objects from the second list are treated as output objects. Additional parameters can be included after the two lists.

Thus, we might describe an inverter by

```
(def (inverter (a) (b))
   (forever a+ b- a- b+))
```

During evaluation of `(inverter in out)`, events on `in` will be treated as input events, and events on `out` will be treated as output events. We will discuss how such treatment differs in later chapters. This inverter assumes that the input wire starts out at low potential (because the first event is a rising edge) and the output wire starts out at high potential. We use the `wire-init-set!` operation to enforce this:

`(wire-init-set!` w v`)` Sets the initial value field of the wire w to v, if it does not already have a value. If it does have a value, then that value and v are compared, and if they are different, an error is flagged.

We also a function `wire-init-known?` that tests to see if the initial value of a wire has already been set:

`(wire-init-known?` w`)` Returns `1` if the wire already has an initial value, and `0` otherwise.

We can obtain the initial value of a wire with `wire-init`:

`(wire-init` w`)` Returns the initial value of the wire if it has one; otherwise, it flags an error.

We also define the `if` and `when` special forms:

`(if` *cond exp1 exp2*`)` Evaluates the *cond* expression. If the expression is simple, numeric, and non-zero, then it evaluates the *exp1* expression and returns that result. If the expression is simple, numeric, and zero, it evaluates the *exp2* expression and returns that result. If the expression is not numeric, then we construct a Petri net version of `if`; we will discuss how this is done in chapter 8.

`(when` *cond exp*`)` This macro expands to `(if` *cond* `(begin .` *exp*`) ())`.

Now we can easily write a macro for the inverter that checks to see if the initial value of the input wire has been set. If it has not, it sets it to zero. In any case, it uses the appropriate model for the initial value and sets the initial value of the output wire to the inverse of its input wire:

```
(def (inverter (a) (b))
   (when (not (wire-init-known a))
      (wire-init-set! a 0))
   (wire-init-set! b (not (wire-init a)))
   (if (wire-init a)
      (forever a- b+ a+ b-)
      (forever a+ b- a- b+)))
```

This type of scheme can be useful when designing a simple model for a circuit element and allowing it to handle all possible combinations of initial conditions, as well as properly propagating those initial conditions to internal and output wires as needed. If the state of the circuit element is not entirely determined by initial values on wires, we may add an additional parameter to indicate that additional state:

```
(def (flip-flop (d clk) (q qbar) init)
   . . .
```

As an alternative, we could also read the initial value from the output wires `q` or `qbar`.

3.9 State Machines

Expressing finite state machines can always be done with a flat regular expression. But this regular expression is often much more complicated and has many more states than the simple finite state machine it is based on. Consider our four-state, two-input, infinitely fast and-gate, as described in chapter 2. The regular expression for all infinite runs of this net is:

$$((a{+}a{-}) \mid (b{+}b{-}) \mid ((a{+}\{b+, c+\}) \mid (b{+}\{a+, c+\}))$$
$$((\{b-, c-\}\{b+, c+\}) \mid (\{a-, c-\}\{a+, c+\}))^*$$
$$((\{b-, c-\}a-) \mid (\{a-, c-\}b-)))^*.$$

We may also use labeled `let` to construct more complicated finite state machines, since the labeled places can be used as targets of branches. However, the scope of the place identifier within a labeled `let` is just the body of that `let`; this allows us to construct any finite state machine that can be decomposed into a tree rooted at the initial state and augmented with backwards edges restricted to the ancestors of the source node. This is still not sufficient to give a concise description of our desired finite state machine.

Engineers often use a simple state transition diagram to represent finite state machines. Our and-gate would look like:

state	action	next
00	$a+$	10
00	$b+$	01
01	$a+, c+$	11
01	$b-$	00
10	$a-$	00
10	$b+, c+$	11
11	$a-, c-$	01
11	$b-, c-$	10

We would like to be able to use a similar description in our language. The `sm` (short for state machine) construct provides this facility. It is similar to a labeled `let`. The `sm` construct indicates that all values bound during net evaluation must be known at evaluation time. In addition, rather than associating a single place with the label, we associate a different place for every combination of input values.

When we initially encounter a labeled `let`, we do not know if all of the expressions in every branch that leads back to that labeled let will have all of the values known. In other words, the expression

```
(let for ((i 0))
   (when (< i 10)
      a+ a- (for (+ i 1))))
```

might describe a data flow machine that counts to ten, generating a pulse on *a* each time, or it might describe an eleven-state state machine, with a pulse on *a* generated between each state. With a labeled `let`, we always assume the former; if there are any identifiers bound within a labeled `let`, we always treat it as describing a data flow construction. When we want partial evaluation, we use `sm` instead.

(`sm` *label* ((*id0 exp0*) (*id1 exp1*) ... (*idn expn*)) . *body*) Constructs a finite state machine, with the *id0* through *idn* as state variables with initial values *exp0* through *expn*. An association table between assignments of values to these variables and places is constructed with an initial element of the initial values of the variables and the current place. Each element of this association table also has a flag indicating whether it has been expanded or not. Then, as long as there are elements of the association table that have not been expanded, a single unexpanded element is chosen, and the current environment is extended by binding *id0* through *idn* with the state values from that element. In this context, *body* is evaluated. Every use of *label* seen in *body* is evaluated by treating the arguments as a state vector and looking up that state vector in the association table. If no such state vector is found, a new one is created. In any case, the current state is merged to the associated place from the association table.

Note how this differs from, for instance,

```
(def (clock? () (clk) s)
   (if s
      (begin clk- (clock? clk 0))
      (begin clk+ (clock? clk 1))))
```

Evaluation of this will never terminate; even though the identifier `s` is bound on each call to `clock?`, there is no association between multiple calls of `clock?` with the same state vector. The proper way to do this is

```
(def (clock () (clk))
   (sm loop ((s (wire-init clk)))
      (if s
         (begin clk- (loop 0))
         (begin clk+ (loop 1)))))
```

Of course, this is equivalent (assuming an initial value of 1 and omitting the header) to `(forever clk- clk+)`.

For state machines, the `guard` construct is useful; this is a combination of `choice` and `if`. Indeed, `choice` is just a macro that is evaluated in terms of `guard`.

`(guard ((`*b1 v1 . bod1*`) (`*b2 v2 . bod2*`) ... (`*bn vn . bodn*`)))` A generalized guard construct. First, all expressions *b1* ... *bn* (known as Boolean conditions, or guards) are evaluated. For all of those that evaluate to a numeric one, the corresponding *vi . bodi* is interpreted with an implicit `begin`. We extend this later for the case where the guards may not be known at compile time, in which case the *v*'s will take on additional meaning.

Now we can easily describe our fast and-gate. Notice how it looks just like the state transition table given earlier.

```
(def (fast-and (a b) (c))
   (wire-init-set! c (and (wire-init a) (wire-init b)))
   (sm loop ((s (+ (* 10 (wire-init a)) (wire-init b))))
      (guard
         ((= s 00) a+      (loop 10))
         ((= s 00) b+      (loop 01))
         ((= s 01) a+ :: c+ (loop 11))
         ((= s 01) b-      (loop 00))
         ((= s 10) a-      (loop 00))
         ((= s 10) b+ :: c+ (loop 11))
         ((= s 11) a- :: c- (loop 01))
         ((= s 11) b- :: c- (loop 10)))))
```

We may also write it as follows, allowing easier extension to an arbitrary number of inputs:

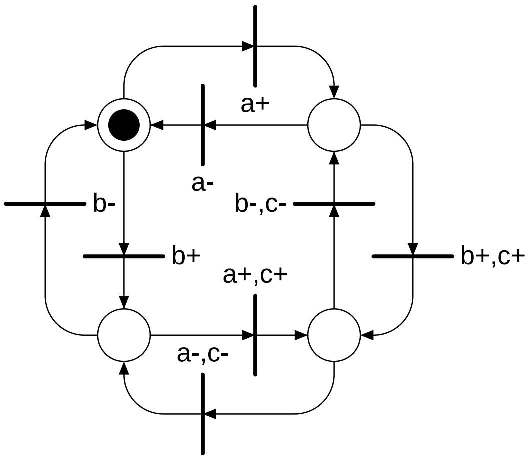


Figure 27. A fast and-gate.

```
(def (fast-and (a b) (c))
   (wire-init-set! c (and (wire-init a) (wire-init b)))
   (sm loop ((av (wire-init a)) (bv (wire-init b)))
      (guard
         ((= av 0) a+ (when (= bv 1) :: c+) (loop 1 bv))
         ((= av 1) a- (when (= bv 1) :: c-) (loop 0 bv))
         ((= bv 0) b+ (when (= av 1) :: c+) (loop av 1))
         ((= bv 1) b- (when (= av 1) :: c+) (loop av 0)))))
```

The resulting net is shown in figure 27. In general, for such combinational elements, we provide a macro that allows us to declare this with just `(sm-ckt (fast-and (a b) (c)) (and a b))`:

`(sm-ckt (`*name* `(`*in-wires*`) (`*out-wire*`))` *exp*`)` Declares an infinitely fast combinational element, with name *name*, computing the function as given by *exp*. The circuit is built using `sm` and `guard` and added to the global environment.

Using `sm`, it is easy to see how to generate any arbitrary orbital net with just a single token flowing through it; this is the state machine subclass of Petri nets. Each place is assigned a unique integer, and each transition is given a single line in the `sm` statement.

Chapter 4

Semantics, Composition and Receptiveness

Mad world! mad kings! mad composition! —Shakespeare, 1564–1616

In this chapter we show how to combine finite state machines to form complex specifications and concurrent systems.

We start by defining the *orbital property* of orbital nets. The orbital property implies safety, but it has additional qualities that lead to more efficient simulation and structural analysis. Then, to motivate composition, we present the behavioral semantics for orbital nets. We discuss groups, which control the synchronization of actions during composition. We present both structural and behavioral definitions of composition, including epsilon places, a syntactic shorthand for synchronization. We discuss the difficulties that arise when synchronizing transitions that are labeled with multiple actions, and show how these problems are resolved. Our solution allows us to consider trace structures of sequences, rather than being concerned with the more complex powerset trace structures. Finally, we define receptiveness and give a structural construction and a syntactic shorthand that allows us to add receptiveness to a non-receptive model.

Composition serves two related purposes in Orbits. The first is to determine the behavior of a complex machine by combining the behaviors of the separate elements comprising that machine. The second purpose is to describe complex behaviors by breaking them down into different aspects. A general difference between these two purposes is that in the first case, we compose receptive models; in the second, we compose non-receptive models.

To illustrate the second use, let us revisit the arbiter with reject that we presented in chapter two. Recall that the `par` statement is the only statement that increases the number of tokens in our nets. Each token represents a stream of execution or control flow. Therefore, `par` is used to indicate that two state machines—each running an infinite loop—execute in parallel. When we introduce synchronization groups and composition, we will see how these finite state machines synchronize on some common set of actions. We use this functionality to describe the arbiter below.

There are two parts to the behavior of this arbiter: the protocol that is used to communicate with each client, and the property that the two clients cannot simultaneously have the resource. Any behavior that satisfies both of these requirements is legal behavior of the arbiter. The entire specification is given as

```
(def (arb-rej (r1 r2) (a1 a2 n1 n2))
   (par
      (process r1+ (choice (a1+ r1- a1-) (n1+ r1- n1-))
      (process r2+ (choice (a2+ r2- a2-) (n2+ r2- n2-))
      (process (choice (a1+) (a2+)) ; give away the token
         (let loop ()  ; as long as the token is gone, nak
            (choice (n1+ (loop)) (n2+ (loop)) (a1-) (a2-))))))
```

Compare this description with the state machine description in figure 5 in chapter two. For now, just read `forever` for `process`. Informally, if a particular event occurs in one finite state machine, and it is mentioned in another finite state machine, then the event must occur in both finite state machines simultaneously. For this example, the only common events among the three finite state machines are $a1-$, $a2-$, $a1+$, $a2+$, $n1+$, and $n2+$. Since each of these appear only once in each finite state machine in which they occur, we merge the transitions that have the same event label, leading to the Petri net shown in figure 4. We will cover these ideas more formally later in the chapter.

4.1 Orbits

Before we discuss composition, we present some structural properties of the orbital net constructed by the evaluation process of the previous chapter. The primary such property is the *orbital property*. Informally, an *orbit* is a set of places through which a single token circulates. The orbital property states that for all reachable markings of the net, no two places from the same orbit will have a token simultaneously. This is a stronger property than safety.

In general, during construction of the net, there is a current orbit and a current orbit increment, both represented by integers (written o and δ, respectively). We treat these two numbers collectively as the orbital pair. The function of the orbital increment is to ensure that the same orbital number is not assigned within two different concurrent branches of the same `par` statement, which might contain any number of nested `par` statements. Every

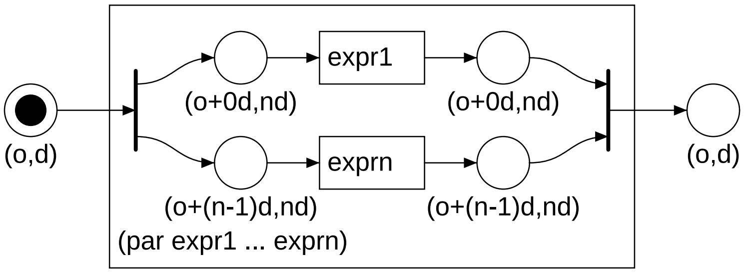


Figure 28. The orbital pair computation for `par`.

place created is annotated by an orbital pair. The only operator in our language that affect the orbit numbers is the `par` operation, which creates new orbits.

The initial token in the net (there is only one) is assigned orbital number 0 and orbital increment 1. When a `par` statement is encountered, the first subexpression is arbitrarily chosen to continue the current orbit, and new orbits are created by incrementing the orbital number by the orbital increment for each of the additional subexpressions. The orbital increment for the new places is generated by multiplying the original orbital increment by the number of subexpressions. Thus, if the current orbital number and increment were o and δ at the beginning of the `par` statement, then during evaluation of subexpression exp_i (assuming we number them starting at zero and there are n of them), the orbital number will be $o + i\delta$, and the orbital increment will be $n\delta$. Every nested `par` statement repeats the process. At the end of each `par` statement, for the place after the end transition (if any), the orbital number and increment are restored. This computation is illustrated in figure 28.

The purpose of this construction, and in particular the orbital increment, is to ensure that no two places with the same orbital number can be marked with a token at the same time. During evaluation of an expression with an initial orbital pair of (o, δ), each place created will have an orbital number of $o + m\delta$ for some nonnegative integer m. Because of the way we generate the orbital increments, these are guaranteed to be distinct in each branch of every `par`.

The place merge operation that is implicit in loops, `choice`, and `guard` statements is restricted in order to ensure the orbital property: places that are merged must have the same orbital pair. In addition, the place label is only accessible within the body of a subexpression, so control cannot go outside that subexpression by using that label. These labels cannot be returned as values. Thus, control cannot branch from a subexpression of a `par` statement to a place (label) before the `par` statement. For `choice` and `guard`, the final places of the subexpressions that are merged are of the same orbital pair by induction, because they have the same initial places. Thus, the only operator that affects orbits is the `par` statement.

We will now prove the orbital property based on the way the nets are constructed. We will refer to the net of an expression as the orbital net that is built when that expression is evaluated. We need a pair of lemmas, after which we show that the second lemma implies the orbital property.

Lemma 4.1. The final place, if any, of the net of an expression has the same orbital pair as the initial place.

Proof. We prove this by structural induction on expressions. This is true of the nets created by all primitive operations but `par` because they do not affect the current orbital pair. The lemma is also true of a `par` statement where no subexpression terminates, as there is then no final place. Finally, if all subexpressions of a `par` statement terminate, then the final place is directly assigned the same orbital pair as the initial place of the `par` statement. ∎

Lemma 4.2. For every reachable marking of the net, every two marked places p and p' have orbital pairs (o, δ) and (o', δ'), respectively, that satisfy the inequality

$$o \not\equiv o' \pmod{\gcd(\delta, \delta')}.$$

Proof. We prove this by induction on prefix-closed firing sequences of the net. This property holds when only the initial place of orbital pair $(0, 1)$ marked. Let there be some reachable marking that violates this property. Select the shortest firing sequence from Q that results in this marking; it must be the final transition fired that causes the property to be violated. We will consider this sequence only for the effects on the set of orbital pairs of places containing tokens, rather than the places themselves.

There are three types of transitions that are constructed in our nets. The first kind, created by `annotate` and the place merge operation, has a single input and output place, and the orbital pairs of the input and output places are identical. Thus, our final transition cannot be one of these. The other two types of transitions are those created for the beginning and the end of a `par` statement.

Let us assume that the transition is one that begins a `par` statement. Each such beginning transition consumes a token from a place of orbital pair (o, δ). If the property held before this transition fired, then there was no token in any other place with orbital pair (o', δ') such that $o \equiv o' \pmod{\gcd(\delta, \delta')}$. The output places of the transition are all of the form $(o + i\delta, n\delta)$. Let us assume there is one such token that violates the property; that is, there exists an i such that $o + i\delta \equiv o' \pmod{\gcd(n\delta, \delta')}$. Since $\gcd(\delta, \delta')$ divides evenly into $\gcd(n\delta, \delta')$, this means that $o + i\delta \equiv o' \pmod{\gcd(\delta, \delta')}$. In addition, $\gcd(\delta, \delta')$ also divides evenly into δ, so this means that $o \equiv o' \pmod{\gcd(\delta, \delta')}$. This contradicts our assumption that the lemma initially held, so the transition cannot be created from the beginning of the `par` construction.

The only remaining type of transition is one that ends a `par` statement. Since the initial place and final place of each subexpression of the `par` statement have the same orbital pair, the ending transition of the `par` statement has precisely the reverse effect of that of the beginning transition of the `par`. Let us assume that the new place marked has orbital pair (o, δ) and that there is some other marked place with orbital pair (o', δ') such that $o \equiv o' \pmod{\gcd(\delta, \delta')}$. The n places unmarked by the firing of this transition have orbital pairs of the form $(o + i\delta, n\delta)$. If the orbital property held initially, then for each $0 \leq i < n$, $o + i\delta \not\equiv o' \pmod{\gcd(n\delta, \delta')}$. This excludes all possible remainders of o' divided by $n\delta$ that are divisible by δ, so $o \not\equiv o' \pmod{\gcd(\delta, \delta')}$, which contradicts our assumption. Thus, the transition cannot be one created by the ending of a `par` statement. We have eliminated all types of transitions in the orbital net, so no firing sequence that violates our lemma can exist.

Since initial marking propagation has precisely the same effect of firing transitions that can only be fired once, this operation does not affect the lemma. In addition, the firing of transitions created by the removal of `::` places (by merging the input and output transitions) cannot violate the lemma property, because firing such a transition is identical to firing a step consisting of the two original transitions.

When we consider composition and synchronization of our nets, we will introduce merge operations on transitions that have the same effect as the firing of two transitions simultaneously. If firing the transitions individually cannot violate the orbital property, neither can firing the transitions simultaneously, so the lemma remains true under such a transition merge operation. ∎

With this lemma, it is trivial to prove the orbital property.

Theorem 4.1. *The orbital property.* In any orbital net constructed by translation from our language, there is no reachable marking such that two marked places have the same orbital number.

Proof. By lemma 4.2, every pair of marked places p and p' in every reachable marking satisfies $o \not\equiv o' \pmod{\gcd(\delta, \delta')}$. This implies that $o \neq o'$. ∎

The orbital increments are only used during construction to create new orbital numbers and to ensure that branches are well formed. After the net is fully constructed, the orbital increment is no longer necessary and can be eliminated. In addition, the orbits are renumbered by assigning successive integers starting with 0, since the orbit numbers generated by the construction process may get large. This benefits various algorithms by allowing them to use a fast and dense array to store information on an orbital basis.

After the net is constructed, each transition is associated with the union of the set of orbits represented by its input and output places. When two transitions are merged through the use of :: or composition, the set of orbits of the resultant transition is the union of the sets of orbits of the original two transitions.

The orbital property can be used in many ways. For instance, in representing the state of the net with a bit vector, knowing that at most one place in an orbit can be marked reduces the size of the bit vector component for one orbit from n bits to $\lceil \log_2 n + 1 \rceil$ bits. Many algorithms can also benefit from this property. For instance, when implementing the stubborn set method [50], a stubborn set of orbits rather than a stubborn set of transitions can be found. This does not appear to decrease the state space reduction of the stubborn set method. Since there are typically many more transitions than orbits, finding stubborn sets tends to be much faster.

Other techniques for determining an appropriate orbital partitioning are possible. Since our nets are safe, a technique that iterates over the net structure, identifying and labeling safe cycles with one or zero tokens will also find orbits that satisfy the orbital property.

4.2 Behavioral Semantics

Before we introduce composition, we provide a formal definition of behavior for circuit elements, so that we can show that our structural composition operation corresponds to the behavioral composition operation. Also, in order to use orbital nets to model circuits, we need an interpretation of the nets in terms of circuit behavior. There are many formalisms that can be used for this; we use the trace theory of David Dill, which has been specifically developed for circuits and their analysis. The ability to use these techniques on our representation illustrates its effectiveness.

In the flavor of trace theory we use, behavior is defined by a prefix-closed trace structure, which is a four-tuple (I, O, S, F). I is the alphabet of input actions and O is the alphabet of output actions; these sets are disjoint, and their union is referred to as Σ. S and F are the *success* and *failure* set, respectively; each is a set of sequences composed of sets of events from Σ: $S \subset (2^{\Sigma})^*$, $F \subset (2^{\Sigma})^*$. S and F are not necessarily disjoint; their union is L, the *possible* set. The combined set L is prefix-closed; every prefix of a member of L is also in L. The success set is also prefix-closed, although the failure set is not.

Informally, the *success* set is the set of action sequences that correspond to good runs of the circuit. The *failure* set is a set of action sequences that have, at some point, a bad event—an input that some model could not handle, or an incorrect output, or, for some models, a hazard.

Each transition in an orbital net is labeled by zero or more actions from the set Σ. We refer to an occurrence of one of these actions as an event. We define a function *lab* that returns the set of actions that labels a particular transition, and extend it to function over steps, in which case it might return a set, and over firing sequences, in which case it returns a sequence of subsets of Σ. This function also removes any transitions that are not labeled, so the resulting sequence might be shorter than the original one. We also define the restriction function $|_P$ on sets or sequences that removes all elements from the set or sequence that are not in P, and we extend it to sequences of sets so that it operates on each sequence element independently, then removes any sequence elements that map to the empty set.

We define a function *filter* that takes a set and a predicate and returns that subset of the set consisting of all members that satisfy the predicate.

Given an orbital net N and disjoint subsets I and O of Σ, we now define the trace structure of N as

$$(I, O, lab(filter(Q, \neg fail)) \mid_{I \cup O}, lab(filter(Q, fail)) \mid_{I \cup O} (2^{\Sigma} - \{\phi\})^{*})$$

We extend our failure set with all suffixes of a failure; once a model has failed, we declare that it can execute any actions, even if the net we are using to describe our circuit does not allow those actions. The algorithms we will be using for verification and simulation declare the failure of any component or the environment to be a global failure; once a failure has occurred, we are generally not interested in any subsequent behavior. It also allows us to use a simple structural definition of a failure; anytime a place is marked that has no output transitions, or no additional transitions are enabled, a failure has occurred. As we develop our representation, we will also describe constructions or extensions to Petri nets that directly model Dill's concepts of composition and receptiveness.

The trace structure of an orbital net as defined above does not satisfy some requirements of a trace structure as defined by Dill; one of these properties is receptiveness. The previous trace structure is a powerset trace structure, which would yield a stronger conformance relation than we want; we show later in this chapter how we consider only simple sequence trace structures if we meet a specific set of conditions.

The labels attached to the transitions of an orbital net have no role in the execution of that net other than as useful annotations when describing execution sequences to the human designer. The labels are important for some operations, such as composing two nets or synthesizing hardware, but the net itself represents the relationships between events; the labels do not affect the analysis.

4.3 Synchronization Groups

Composition of labeled transition systems has almost always been done by synchronizing like-labeled transitions in each state machine. That is, in order for any action to occur in the composite machine, it must occur in every state machine that might possibly cause that action. This form of composition is powerful and satisfies many important algebraic properties, but it makes it difficult to describe certain situations, as in systems where resources are shared by a number of entities.

Consider describing the instruction fetch and data fetch portions of a microprocessor. These two processes run in parallel, with their own control flow, perhaps checking separate caches and feeding data to different portions of the chip. Under certain conditions, they require data from the data bus, so the signals controlling the data bus are part of each of their state machines. Yet, not all actions on the data bus are described by either state machine, but rather the union of the two. In addition, they must arbitrate for use of the data bus.

The fetch phase of the microprocessor might be reduced to something like

```
(forever checkIcache+
   (choice (inIcache+) (notinIcache+ busreadreq+ busreadack+)))
```

while the data fetch looks like

```
(forever checkDcache+
   (choice (inDcache+)
      (notinDcache+ (if read (begin busreadreq+ busreadack+)
                             (begin buswritereq+ buswriteack+)))))
```

When these two descriptions are composed, the events `busreadreq+` and `busreadack+` must not be synchronized, since each state machine must be able to perform these actions independently. The `complete` keyword declares *synchronization groups* which control how synchronization is performed.

`(complete (`*ev0 ev1* ...`) .` *body*`)` Specifies that all state machines built by the evaluation of `body` shall be synchronized with all other state machines on events given in the event list. Objects may be part of the event list, in which case all possible events on these objects will be used. When the event list is empty, it is treated as though all possible events were listed.

Thus, synchronization is performed on all actions shared by two state machines if both state machines are complete:

```
(par
   (complete () ... state-machine1 ...)
   (complete () ... state-machine2 ...))
```

This is the most common type of synchronization. We define the keyword `process` because this is so common:

`(process` . *body*`)` This is a macro abbreviation for `(complete () (forever .` *body*`))`, since that is the most common type of infinite loop.

In the example of the microprocessor, on the other hand, we do not want to perform synchronization on the `busreadreq` and `busreadack` wires. We list, in a `complete` statement, the events that we do wish synchronization to take place on for each of the two state machines:

```
(par
   (complete (checkIcache notinIcache)
      (forever checkIcache+
         (choice (inIcache+) (notinIcache+ busreadreq+ busreadack+))))
   (complete (checkDcache inDcache buswritereq buswriteack)
      (forever checkDcache+
         (choice (inDcache+)
            (notinDcache+ (if read (begin busreadreq+ busreadack+)
                                   (begin buswritereq+ buswriteack+)))))))
```

In this case, the two state machines will synchronize on all wires but `busreadreq` and `busreadack`. Since they have no wires in common, we may use instead

```
(par
   (forever checkIcache+
      (choice (inIcache+) (notinIcache+ busreadreq+ busreadack+)))
   (forever checkDcache+
      (choice (inDcache+)
         (notinDcache+ (if read (begin busreadreq+ busreadack+)
                                (begin buswritereq+ buswriteack+))))))
```

To state that the two machines together synchronize with their environment on all of their actions, we enclose the `par` statement in a `complete` statement with an event list of `()` to indicate all events.

No synchronization will be performed if there are no `complete` statements in the entire expanded expression.

Each `complete` statement introduces a new synchronization group on a particular class of actions. Common events within the same synchronization group do not synchronize with each other, but they do synchronize with every other synchronization group that contains that particular event. In the statement

```
(par
   (complete (a)
      (forever a+ b+ a+ b+)) ; first process
   (complete (a b)
      (forever a+ b+ a+ b+)) ; second process
   (forever a+ b+ a+ b+))    ; third process
```

no pair of $a+$'s or $b+$'s on the same line will synchronize with each other, since such pairs are members of the same synchronization group. Each of the three processes belongs to a separate synchronization group on events on wire a, since the first two processes declare a new group on a and the last inherits the synchronization group of the overall `par` statement. The first and third processes are members of the same synchronization group on b, so synchronization will not take place between these two processes on b. The second process is itself a synchronization group on b, so every occurrence of $b+$ in the second group will be synchronized with one occurrence of $b+$ in either the first or the third process, but not both.

For any subexpression, and also for the net of that subexpression, certain actions may occur outside the scope of a `complete` on that action; we refer to the set of all such actions as the *incomplete* set for that expression. Other actions may occur within the scope of a `complete`; this is the *complete* set of actions. These two sets are not necessarily disjoint.

With `complete`, it is syntactically possible to declare synchronization groups that split a state machine for a particular action. For example,

```
(begin (complete (a) a+) a+)
```

is such an expression. Any such net will never perform that action. This condition is easy to check; any pair of subexpressions of a `choice`, `guard`, or `begin` must have disjoint complete sets of actions.

Arbitrary groups significantly complicate trace structure analysis of the composition of nets, because they make it possible for step firings of transitions to contain multiple occurrences of the same action. Allowing this might be useful. For instance, if there are multiple clients and multiple servers, two identical interactions might be concurrently enabled and executed simultaneously. But for the remainder of this dissertation, we will consider trace structure analysis only of nets that do not allow concurrent execution of the same action. While intermediate stages in the composition of a system may allow concurrent execution of an action, the final composite net must not. This requirement is easily enforced; for each action, some complete group must serialize all occurrences of that action. It is also easily checked. Any complete simple state machine without any `par` statements automatically satisfies this restriction for all actions in its alphabet. The presence of terminating `par` statements is also allowed; we insist that all subexpressions of a terminating `par` statement have a disjoint set of actions, so that no action can be concurrently enabled in two branches of any `par` statement.

Composition of incomplete components is primarily useful in decomposing specifications; the composite specification is generally complete. For the circuit trace structure analysis, we require that all components be complete in all actions in their alphabet. This is necessary in order to satisfy the properties of a circuit algebra.

4.4 Composition of Trace Structures

Now we will discuss composition a little more formally. Initially, we will discuss it in terms of event traces, and then we will show how to compose orbital net models to get equivalent behavior.

Composition of trace structures follows classic trace theory [15] on individual events. If we have two nets whose behaviors are represented by two trace structures, the composition of the two nets must have behavior represented by the composition of the trace structures. We must extend trace structure theory to apply to sets of simultaneous actions.

First, we need an inverse restriction operation. The operation $|_P^{-1}$, given a set of actions P, operates on a set of sequences s and returns the set of all sequences that can be constructed by adding any number of actions from P at any position in any sequence from s. If s is a set of sequences of sets of actions, then in addition to the interleavings, it also possibly adds any subset from P to the existing sequence elements. More formally, $\alpha \in s\,|_P^{-1}$ if and only if $\alpha\,|_P \in s$.

We extend the inverse restriction operation to trace structures as follows:

$$TS\,|_P^{-1} \equiv (I \cup P, O, S\,|_P^{-1}, F\,|_P^{-1}).$$

We define the intersection of two trace structures that share the same alphabet and have disjoint output sets as

$$TS_1 \cap TS_2 \equiv (I_1 \cap I_2, O_1 \cup O_2, S_1 \cap S_2, (F_1 \cap P_2) \cup (P_1 \cap F_2))$$

Finally, with these two definitions, we define the composition of two trace structures as

$$TS_1 \parallel TS_2 \equiv TS_1 \mid^{-1}_{A_2-A_1} \cap TS_2 \mid^{-1}_{A_1-A_2}$$

This definition is the same used by Dill [15] and applies to our powerset trace structures as well as to interleaving trace structures.

Consider a trace structure with an empty failure set and with a success set defined by all prefixes of $(\{a+\}, \{b+\})^*$:

$$TS_1 = (\{a\}, \{b\}, \mathit{pref}((\{a+\}\{b+\})^*), \phi)$$

This might model a non-receptive two-phase buffer with input a and output b. We illustrate trace structure composition by composing this with an identical two-phase buffer, but with input b and output c:

$$TS_2 = (\{b\}, \{c\}, \mathit{pref}((\{b+\}\{c+\})^*), \phi)$$

We adopt the shorthand that b in an alphabet stands for all events that can occur on b. First, we extend the alphabets of the two trace structures to $\{a, b, c\}$ using the inverse restriction operator:

$$TS_1 \mid^{-1}_{\{c\}} = (\{a, c\}, \{b\}, \mathit{pref}((\{c+\}^*(\{a+\} \mid \{a+, c+\}), \{c+\}^*(\{b+\} \mid \{b+, c+\}))^*), \phi)$$
$$TS_2 \mid^{-1}_{\{a\}} = (\{a, b\}, \{c\}, \mathit{pref}((\{a+\}^*, (\{b+\} \mid \{a+, b+\}), \{a+\}^*, (\{c+\} \mid \{a+, c+\}))^*), \phi)$$

We take the intersection of these two trace structures to calculate the total composite trace structure:

$$TS_1 \parallel TS_2 = (\{a\}, \{b, c\}, \mathit{pref}(\{a+\}(\{b+\}((\{a+\}\{c+\}) \mid (\{c+\}\{a+\}) \mid \{a+, c+\}))^*), \phi)$$

If the wire b is not connected to any other component, we can hide it by restricting this trace structure to $\{a, c\}$:

$$(TS_1 \parallel TS_2) \mid_{\{a,c\}} = (\{a\}, \{c\}, \mathit{pref}(\{a+\}((\{a+\}\{c+\}) \mid (\{c+\}\{a+\}) \mid \{a+, c+\})^*), \phi)$$

4.5 Composition of Nets by Merging Transitions

Trace structures define the behavior of a net, but as infinite sets, they tend to be much too large to manipulate directly. We provide a structural composition operation that builds a composite net with the composed trace structure as its behavior.

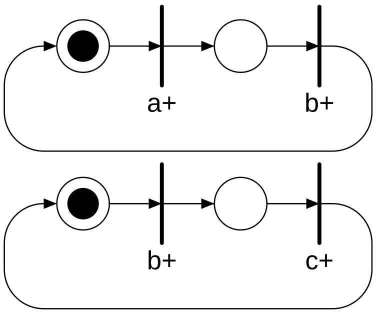


Figure 29. Two simple two-phase buffers.

Composition of the trace structure behavior is based on simultaneous execution, or synchronization, of actions from different groups. In general, if an action occurs in a particular group, it must occur in all groups that include that action; conversely, no action is enabled to occur unless it is enabled to occur in all groups. We perform this structurally by merging like-labeled transitions such that each final transition is a composite of one transition from each group containing a particular action.

Synchronization is straightforward if each group only contains a single transition labeled with the relevant action; all of these transitions are merged into a single transition. For instance, consider the trace structures we composed in the previous section. An orbital net description of the first can be given (assuming the wires have been declared) by

```
(forever a+ b+)
```

yielding the top net shown in figure 29. The second trace structure can be described similarly, yielding a similar net.
To compose these two buffers, we would declare that each is complete in all actions:

```
(par
   (process a+ b+)
   (process b+ c+))
```

Behaviorally, an action can occur only if all conditions for that action in both component machines are satisfied. In our net models, two transitions labeled with the same action must fire concurrently. We can force these two transitions to occur together by merging the transitions, as shown in figure 30. This net generates precisely the regular language given by the composition of the trace structures in the previous section.

If a synchronization group contains multiple occurrences of transitions labeled with a particular action, synchronization is more complicated. For simplicity, we will give this

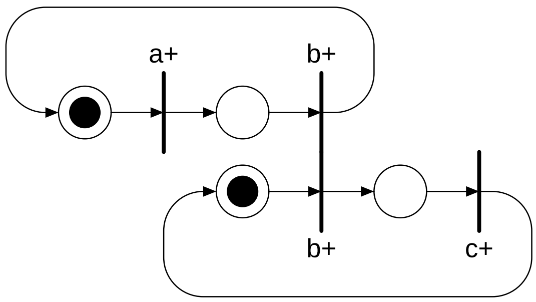


Figure 30. The composition of the two buffers.

construction as though there were only two groups and a single action. The overall synchronization procedure for an entire orbital net is given by:

```
for (a in actions)
   while (|groups(a)| > 1)
      combine-two-groups-on(a)
```

We wish to allow all behaviors where any combination of occurrences of an action from the groups synchronize, so long as each group participates. Since the transitions are the nodes labeled with action occurrences, and since each group is defined over a particular action, a group is essentially a set of transitions. For the composition of two groups, we take the cross product of the two sets, replicating each transition in the pair and merging the two replicated transitions together. We then eliminate the original, unmerged transitions. Replication is done by creating a new transition with the same set of input places, output places, and the same labels as the transition being replicated. It is easy to show that replicating transitions does not change the behavior of the net; there are no additional reachable markings, and there are no additional event sequences in the language of the net. Thus, the individual behavior of the elements being composed is unchanged.

It is easy to demonstrate that this construction corresponds to behavioral composition. Inverse restriction is the same as adding new transitions, labeled with the newly introduced actions, with no input and no output places. These transitions are enabled to occur in every state, and firing them has no effect on the net state. Intersection is implemented by synchronizing like-labeled transitions—an action can occur only if it can occur in both machines.

Because our synchronization procedure considers each action separately, two transitions to be merged might already have been merged. Consider two separate transitions, each labeled $\{a+, b+\}$, and each in separate groups on both events. After synchronizing with respect to $a+$, we have just a single transition, but it is a member of two different groups on $b+$. When we try to synchronize on the $b+$ action, we must note that they are already combined.

Synchronization may connect several concurrent transitions from each state machine. If in one group complete in all actions we have a transition labeled $\{a+, b+, c+\}$, and in another group complete in all actions we have three transitions labeled $\{a+, d+\}$, $\{b+, e+\}$, and $\{c+, f+\}$, then synchronization will merge all of these transitions into a single transition labeled $\{a+, b+, c+, d+, e+, f+\}$.

Not all transitions created by this synchronization process can ever be enabled. In particular, if a merged transition ends up using two transitions from the same group that have an orbit in common, then the resulting transition can never fire, as any two transitions from the same orbit cannot be concurrent. We eliminate such transitions since they have no effect on the behavior of the composite net.

After synchronizing all groups on all actions, we might have transitions with multiple occurrences of the same action from the same group. For instance, merging two transitions labeled $\{a, b\}$, where the a's are from the same group but the b's are from different groups, allows the simultaneous execution of two a events from the firing of the merged transitions. Since we require that all actions be serialized by some group, such a transition can never fire and can be eliminated.

4.6 Epsilon Places

Composition of nets by merging transitions works well when the number of occurrences of each action in the composed nets is small. But if there are a large number of occurrences, the total number of transitions becomes large. For example, our fast and-gate has 2^{n-1} occurrences of each input event; a four-input and-gate will have eight separate occurrences of $a+$. A wire a connected to four and-gates produces 4,096 transitions in the composed net. This leads to inefficiencies in verification, simulation and other operations on the final Petri net, since these operations are often linear (or worse) in the size of the net.

We introduce *epsilon places* as a syntactic shorthand for the synchronization or cross product operation. An epsilon place has a set of input transitions and a set of output transitions. Only one of the input transitions and one of the output transitions can fire at a time, and one input transition and one output transition must occur simultaneously as a single firing. Every epsilon place connected to a transition must be treated in this way. There is no semantic difference between a net with an epsilon place and the same net where the input and output transitions of that epsilon place have been replaced by the merged transitions resulting from the cross product of the input and the output transitions.

For instance, an epsilon place with a single input and single output transition is semantically equivalent to the same net with the epsilon place removed and the input and output transitions merged. An epsilon place with no input transitions will never allow its output transitions to fire, so the epsilon place and the output transitions can be removed from the net without changing its behavior. Since epsilon places are fully symmetrical with respect to input and output transitions, an epsilon place with no output transitions will never allow its input transitions to fire, and both the epsilon place and input transitions can be removed.

More formally, we change the enabledness condition of our Petri nets to take epsilon places into account. No individual transition that has one or more epsilon places as input or output places can fire by itself. A step is enabled only if the step is enabled when ignoring epsilon places, and the step includes, for each epsilon place in the net, either no transitions from either the input or output sets of the place, or exactly one transition from the input set and one transition from the output set of the place.

This more complex enabledness condition is the price we pay for a much smaller net. Conventional Petri net algorithms are easily adapted to nets with epsilon places. The advantage of epsilon places is that an entire class of such combinations can often be quickly characterized (for instance, as not enabled) by examining a single component transition.

An example of groups, synchronization, and epsilon places is given in figure 31. This figure shows an alternative model for our fast 3-input and-gate for signal wires. The four places on the bottom keep track of how many inputs are high; as each input goes high, the single token in the bottom four places rises to the next higher place, and as each input goes low, the single token falls to the next lower place. When all inputs are high, the output finally goes high. This net does not have choice, even though it has places with multiple output transitions for a given place; at any point, only one of the multiple output transitions can be enabled. The places marked with `[]` are epsilon places. With this model, the size of the net is linear in the number of inputs, yet it has exactly the same behavior as the model we presented in chapter 3, with a net size exponential in the number of inputs. This approach can be used to construct models of and, or, not, and exclusive-or gates and their negated forms.

Describing this net requires grouping constructs:

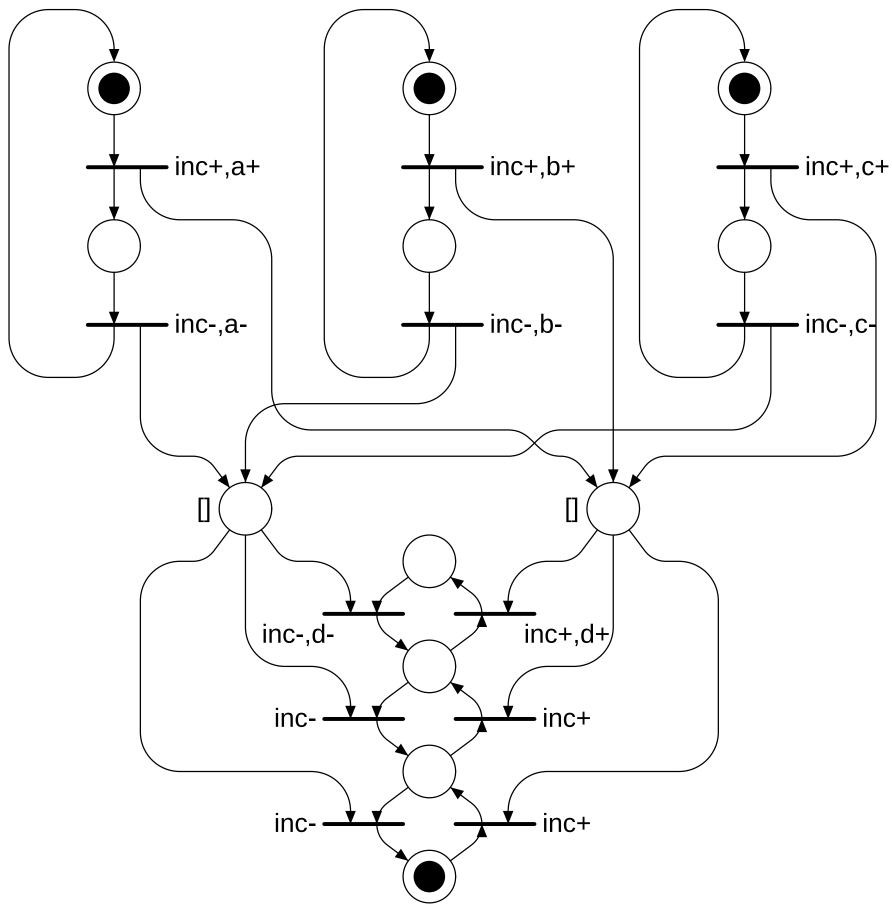


Figure 31. A 3-input and-gate for signal wires.

```
(def (and3 (a b c) (d))
   (with-wires (inc)
      (par
         (complete (a) (forever a+ :: inc+ a- :: inc-))
         (complete (b) (forever b+ :: inc+ b- :: inc-))
         (complete (c) (forever c+ :: inc+ c- :: inc-))
         (complete ()
            (sm loop ((s 0))
               (guard
                  ((and (> s 0) (<= s 3))
                   inc- (when (= s 1) :: d-) (loop (- s 1)))
                  ((and (>= s 0) (< s 3))
                   inc+ (when (= s 2) :: d+) (loop (+ s 1)))))))))
```

The loop that includes $a+$ and $a-$ is complete for the signals on the wire a, but it cannot be complete for the signals on the wire *inc*, because that would preclude events on b and c from affecting our counter.

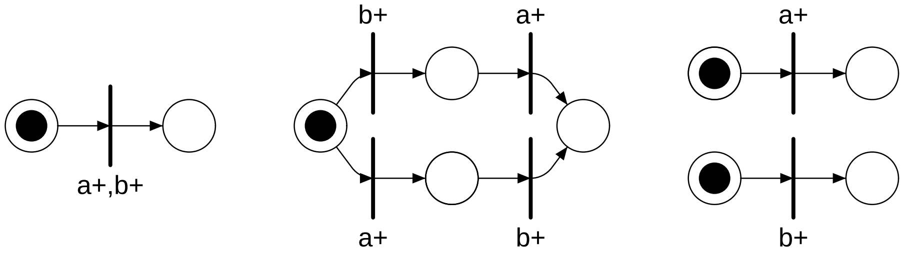


Figure 32. Three nets that allow both $a+$ and $b+$ to occur.

4.7 Interleaved Versus Simultaneous Actions

There is a subtle distinction between allowing two events to occur in either order and allowing them to occur in parallel—in the former case, they cannot occur simultaneously. If we use interleaving semantics, where we only consider sequences of events, rather than sequences of sets of events, this distinction does not arise. But since we require simultaneous events, we must consider this distinction, the problems it causes, and how they may be resolved. In particular, we present a structural check on nets that can avoid unintended deadlock situations when certain models are composed. We also show how this same structural condition allows us to use simple trace structures of events which have better properties for analysis.

We have argued that we must label some transitions with multiple actions, for instance, to separate the function of a gate from its delay characteristics. Allowing the labeling of transitions with multiple actions introduces its own set of problems, however. Consider the three nets pictured in figure 32. All three nets depict the occurrence of actions $a+$ and $b+$. The net on the left insists that they occur simultaneously; the only complete trace that this net allows is $[\{a+, b+\}]$. This type of description might be used if event $a+$ caused event $b+$, and we insist that there be no delay between the two.

The middle net allows $a+$ and $b+$ to occur in any order, but not simultaneously; here, the complete traces allowed are $[\{a+\}, \{b+\}], [\{b+\}, \{a+\}]$. This form of description might be useful if we are trying to construct a state machine where both $a+$ and $b+$ can occur in any order, but either might not occur, or their order might affect the behavior of the net. For instance, the model of our primitive gates uses such a description for all of the inputs.

The net on the right allows $a+$ and $b+$ to occur in any order or simultaneously, so the trace structure is a union of the previous two. This form is used where a is an input to a different circuit component than b is. This net includes the behaviors of the previous two nets.

We cannot compose a net similar to the leftmost net with one similar to the middle net, because there are no traces that are accepted by both nets—if we try to simulate

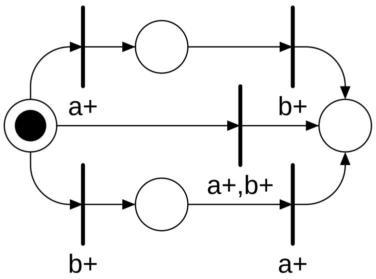


Figure 33. Extending a net with interleaving to also allow simultaneous actions.

the composition, the net will deadlock. Thus, it is possible for us to construct perfectly reasonable Petri nets describing the behavior of some set of circuit components, but not be able to compose them to generate the composite behavior, because of conflict between events that need to be simultaneous in one component, but must be interleaved in another.

One fix extends all nets that allow only interleaving to also allow concurrent actions. Thus, the middle net might be extended to look like the one in figure 33. This net allows the full set of traces accepted by the fully concurrent execution of $a+$ and $b+$. In general, this extension significantly complicates a model; an n-input gate, for instance, would require $2^n - n - 1$ additional transitions from each state to allow all possible simultaneous sets of actions.

On the other hand, use of simultaneous actions is only required in certain specialized places, such as from an input to an output of a fast combinational gate. If we somehow ensure that a conflict between simultaneous occurrence of two actions in one net and interleaved occurrence of the same two actions in another net never happen, we can safely compose the two nets. We therefore develop a structural property on nets that guarantees that this behavioral difficulty never arises.

We must identify all sets of actions that might occur simultaneously in a single transition firing. Those that occur simultaneously due to a step firing do not cause a problem, as we can always fire the elements of step separately in any order and obtain the same total effect. We also identify all actions that might be interleaved. Then, when composing two net elements, we check that there is no pair of actions that might occur simultaneously in one component and occur interleaved in the other.

Consider the fast and-gate, as shown in figure 34. We find all sets of events that might occur simultaneously by examining the labels of the transitions. In this model, the only sets of actions that can occur on a single transition firing are $\{a+\}$, $\{a-\}$, $\{b+\}$, $\{b-\}$, $\{a+, c+\}$, $\{b+, c+\}$, $\{a-, c-\}$, and $\{b-, c-\}$. We refer to this set of sets as the *sim* set.

Calculating the set of events that might be interleaved is more complex. It is sufficient to consider the set of pairs of events that might be in conflict as a conservative approximation

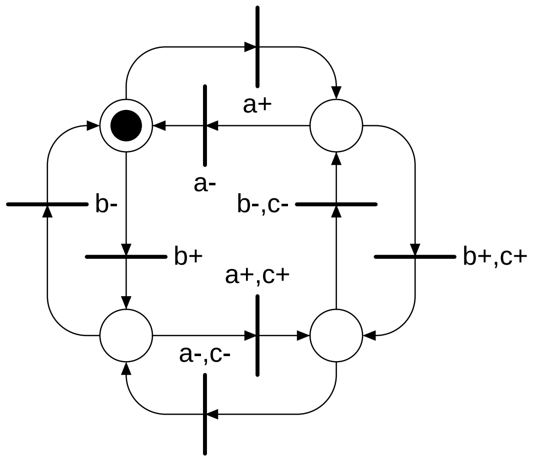


Figure 34. The fast and-gate model.

of those that might be interleaved. This property is calculated by examining the labels of the output transitions from each place in turn. For our fast and-gate, the set of such pairs of events is $\{a+, b+\}$, $\{a+, b-\}$, $\{a-, b+\}$, $\{a-, b-\}$, $\{a-, c+\}$, $\{b-, c+\}$, $\{a-, c-\}$, $\{b-, c-\}$. We refer to this set as the *int* set, since it includes all potentially interleaved pairs of actions.

Behaviorally, we intend the *int* set to be all pairs of actions that, from a given state, might occur in either order, yet cannot occur simultaneously. In terms of trace structures, we what the pair of actions $\{a, b\}$ to be in the interleaved set if there is some α such that $\{\alpha ab, \alpha ba\} \in L\mid_{\{a,b\}}$ but $\{\alpha\{a, b\}\} \notin L\mid_{\{a,b\}}$. This can only happen if the two events are in conflict, although their being in conflict does not guarantee that they will be interleaved.

Similarly, we define the *sim* set as all pairs of actions that, from a given state, might occur simultaneously, but cannot occur interleaved in any order. Thus, we want the set $\{a, b\}$ to be a subset of an element of the *sim* set if there is some α such that $\{\alpha\{a, b\}\} \in L\mid_{\{a,b\}}$ but $\{\alpha ab\} \notin L\mid_{\{a,b\}}$ or $\{\alpha ba\} \notin L\mid_{\{a,b\}}$. This can only happen if the two events label the same transition.

Given such sets, we can compose two nets N and N' without a conflict between interleaved and simultaneous actions if no element of *int* is a subset of any element of sim' and no element of int' is a subset of any element of *sim*.

For a given net, the *int* set might have elements that are subsets of the *sim* set of that net, without causing any problems. This is because we have restricted our nets to not allow any steps to be enabled that have multiple occurrences of an action, and thus the transitions that made those actions part of the *int* set can never be enabled at the same time as those that are part of the *sim* set.

The rename and project operations on our *sim* and *int* sets are straightforward. But the composition operation is slightly more complex. Composition merges transitions according to a set of actions that may occur. When merging nets N_1 and N_2 to form N_3, we calculate

sim_3 and int_3 by building the composite net without epsilon places and then examining the net. But if we wish to use epsilon places, we can compute the sets essentially as though we were merging transitions, but without performing the merge. Specifically, we iterate through the actions, just as we did for synchronization. We locate all *sim* elements with that action in each component net, and compute the cross product as we did for transitions. When we merge two elements of the component *sim* sets into an element of the composite *sim* set, we also extend the composite *int* set conservatively by calculating all transitions that might be synchronized with a particular transition, and extending the *int* set according to this combination.

We require each component on which we perform trace structure operations to have a *sim* set consisting of only singleton sets. If two such components are composed, the resulting *sim* set also is composed of just singleton sets. We can therefore use a simple interleaving trace structure rather than a powerset trace structure, since for every trace with an element that has more than one action, the trace structure also includes traces with those actions interleaved in arbitrary order.

The requirement that the *sim* set be only singleton actions is naturally satisfied. For instance, every combinational gate created by `sm` or `sm-ckt`, when composed with a delay element where the input and output cannot be simultaneous, and when the internal wire is hidden, has such a *sim* set. We do not consider the trace structure behavior of an infinitely fast and-gate, but only of the composition of the and-gate and some delay.

To show how much simpler trace structures of sequences are, let us review the trace structure calculations for a pair of two-phase buffers, this time considering only simple sequences:

$$TS_1 = (\{a\}, \{b\}, \mathit{pref}((a{+}b{+})^*), \phi)$$
$$TS_2 = (\{b\}, \{c\}, \mathit{pref}((b{+}c{+})^*), \phi)$$
$$TS_1 \mid^{-1}_{\{c\}} = (\{a, c\}, \{b\}, \mathit{pref}(((c{+})^* a{+}(c{+})^* b{+})^*), \phi)$$
$$TS_2 \mid^{-1}_{\{a\}} = (\{a, b\}, \{c\}, \mathit{pref}(((a{+})^* b{+}(a{+})^* c{+})^*), \phi)$$
$$TS_1 \parallel TS_2 = (\{a\}, \{b, c\}, \mathit{pref}(a{+}(b{+}((a{+}c{+}) \mid (c{+}a{+})))^*), \phi)$$
$$(TS_1 \parallel TS_2) \mid_{\{a,c\}} = (\{a\}, \{c\}, \mathit{pref}(a{+}((a{+}c{+}) \mid (c{+}a{+}))^*), \phi)$$

4.8 Receptiveness

A circuit cannot directly control the values applied to its inputs—this observation fundamentally distinguishes the inputs of a circuit from the outputs. Any formalism which

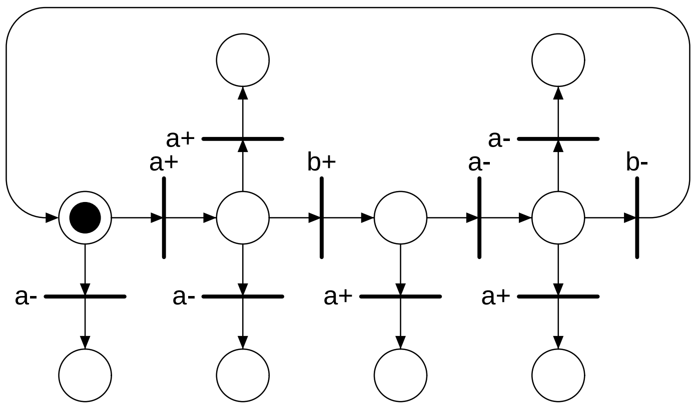


Figure 35. A receptive version of our simple buffer.

purports to allow the modeling or reasoning about circuits must reflect this asymmetry of communication. *Receptiveness* formalizes this for circuit models.

Consider again our basic model of a buffer as given in figure 2. This model describes a buffer with arbitrary delay operating in such a manner that inputs and outputs alternate. When this net is composed with nets representing other circuit elements, it will prevent the occurrence of input events that it is not ready for.

On the other hand, this buffer model is simple because it is not receptive; describing the unsuccessful behaviors of a circuit element often adds significant complexity to the model. By providing a facility for adding receptiveness to a model, we may focus our attention on only the successful behaviors and allow receptiveness to add the necessary failures to our model.

Receptiveness is important outside of modeling circuits; any systems that have a communication channel (such as a wire) where the receiver cannot directly control whether an input is sent or not requires some form of receptiveness.

Receptiveness is related to synchronization of concurrent actions through composition. With synchronization, all relevant elements must agree to perform an event in order for that event to be enabled. For receptiveness, a particular subset of elements, usually of size one, can decide to perform an event, and all other elements must be prepared to cooperate at all times.

For safety properties, it is sufficient that any input action is allowed to occur at any time. With orbital nets, we require that, in every marking reachable in a model net and for each input action, some transition labeled with that action is enabled.

To eliminate potential problems with simultaneous input transitions, we also require that no two inputs be simultaneous. This helps us to derive a structural construction for receptiveness.

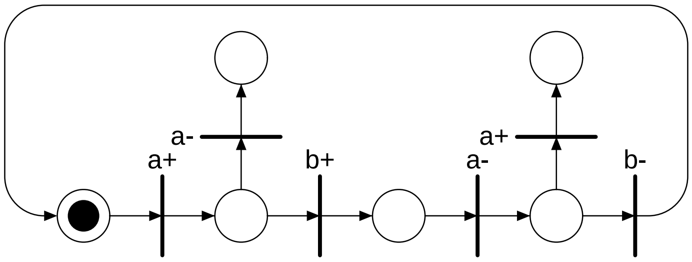


Figure 36. A reduced receptive version of our simple buffer.

It is a simple matter to extend our buffer model to make it receptive by adding the appropriate transitions, as in figure 35. If an input event occurs that does not fall into the sequence $[a+, b+, a-, b-]$, that event is a failure—the environment of the circuit generated an input event that the circuit element, as we have modeled it, was not prepared for. Remember that we define our failure set in terms of transition firing sequences that either leave a place marked with no output transitions or that reach a marking with no enabled transitions, so the new places we add will make the firing of one of the new transitions a failure.

The new places have no output transitions, so their tokens cannot participate in the future behavior of the net. This would seemingly violate receptiveness. However, we have defined the trace structure of a net such that all possible suffixes of a failure are also in the failure set, and thus in the possible set. Because we are not concerned with any behavior after a failure occurs, we do not add additional failure transitions.

Indeed, we do not even need all of the additional transitions pictured. For wires where we model the rising and falling edges separately, we know that these events alternate. If the circuit element generating these events alternates them, then we know that most of these transitions can never be enabled. If some orbit complete in actions on these wires can be separated by transitions labeled with rising and falling actions into two sets, one in which the wire is high and the other in which the wire is low, then we know the actions must alternate. For such a case, we can eliminate almost all of the transitions, leaving only those shown in figure 36 (assuming the correct initial conditions).

Many of our models are implicitly receptive. For instance, our fast and-gate is receptive so long as whatever it is connected to is also receptive. Some inputs are simultaneous with the output, and anything that can prevent an output from occurring can also prevent an input from occurring. But for those models that are not implicitly receptive, it not difficult to make them so.

The construction for adding receptiveness depends on our nets being safe. For the moment, assume that the synchronization is done by merging transitions, rather than using epsilon places. To make a net receptive in a particular action, we identify all transitions labeled with that action. We introduce additional transitions, at least one of which is enabled specifically when none of the existing transitions is enabled.

Because our nets are safe, we can easily construct complement places for any given place. A complement place is one that is marked whenever its corresponding place is not marked and is not marked whenever its corresponding place is. These places are constructed by adding a new place to the net whose input transitions are the output transitions and whose output transitions are the input transitions of the corresponding place. If the original place is initially marked, then the complement place is initially unmarked, and vice versa. It is apparent that, in all reachable markings of the net, the number of tokens in the original place and the complement place sum to one. We only need construct a single complement place for each original place, even if the original place is an input to multiple transitions labeled by the specific action.

Next, we add the failure transitions. We add a transition for every subset α of the set of complement places such that each original transition labeled with the action has at least one place in α as an input place, and each place in α is an input place to some such original transition. We have to keep only the minimal subsets that satisfy this requirement; if both α and β are subsets of the complement places satisfying this requirement, and $\alpha \subset \beta$, we only build a new transition for α. We label each of these new transitions with the action in question and then add a failure place as output.

This construction can generate a complicated net, much like synchronization can greatly increase the size of the net. To help alleviate this problem, we introduce a variant of epsilon places called receptiveness places. These behave almost exactly like epsilon places, except that if an input transition of this receptiveness place is enabled, and no output transition is enabled, then a composite failure transition is enabled consisting of the input transition and a special failure transition introduced specifically for this purpose. The algorithms that handle epsilon places are easily extended to deal with receptiveness places.

Existing algorithms for Petri nets are easily modified to apply to receptive places much like they are for epsilon places. The additional complexity of another style of place is compensated for by the large number of places and transitions that the receptiveness place replaces.

Almost all of our circuit elements are inherently receptive, so we only use receptive places when composing more complex specifications or performing the mirror operation, which we will discuss in the next chapter.

Receptive places are introduced with the `receptive` statement.

`(receptive` *circ*`)` Constructs and returns a receptive version of the circuit specified. The input wires are given new names. A new transition from a new group is allocated for each action on both the new and the old versions of the input wires, and a receptive place is connected between each pair. Composition will later make all of the additional connections necessary.

4.9 Translating Orbital Nets to Text

Orbital nets are sometimes easier to understand and simpler to design in graphical form. In this section, we show how arbitrary safe labeled Petri nets can be translated into our language.

First, each transition in the input net is associated with an identifying action. If a transition is labeled with an action that only occurs once in the entire net, that action can be used. Otherwise, a new wire is allocated and a rising edge on that wire is used as the action associated with that transition.

For each place, a new infinite loop is introduced. The loop must be complete in all actions identifying with its input and output transitions; if none of these identifying actions were allocated in the previous step, or if the actions that do not identify the transitions are not mentioned in this particular loop, then it can be complete in all actions. If the place is initially marked, then the loop consists of a choice of all identifying actions for its output transitions followed by a choice of all identifying actions for its input transitions; otherwise, the choice of input transition actions is followed by a choice of output transition actions. If there is only one input or output action, of course, no choice is necessary. In addition, each non-identifying action labeling a transition must occur in exactly one of the loops introduced for places that connect to that transition.

If the place has any transitions that both consume and restore the token from that place, then these transitions are not listed as part of the input or output set in the normal manner. Instead, they are inserted as an internal loop at the front of the loop if the place is initially marked, or at the end of the loop if the place is not initially marked. This internal loop chooses to fire any of these transitions, or exit and continue the outer loop.

If two loops are identical, one can be deleted.

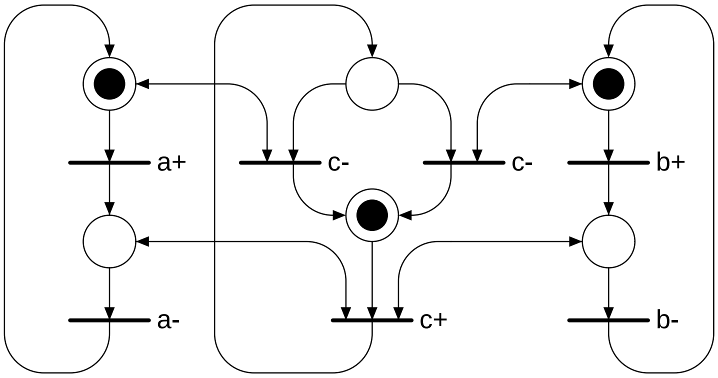


Figure 37. A sample safe Petri net.

For example, the net given in figure 37, taken from chapter two, can be written using this procedure. We construct loops for each place considering the columns of places left to right and top to bottom within each column.

```
(complete ()
   (with-wires (c-/1 c-/2)
      (par
         (process (let loop (choice (c-/1+ (loop)) ())) a+ a-)
         (process a+ (let loop (choice (c+ (loop)) ())) a-)
         (process c+ (choice (c-/1+ :: c-) (c-/2+ :: c-)))
;        (process c+ (choice (c-/1+ :: c-) (c-/2+ :: c-)))
         (process (let loop (choice (c-/2+ (loop)) ())) b+ b-)
         (process b+ (let loop (choice (c+ (loop)) ())) b-))))
```

The orbital net resulting from the translation of this text has the same number of transitions as the original net, but twice as many places. These places can usually be eliminated by a more careful translation that finds the longest possible connected cycles or paths of places having exactly one token in it, and then using a similar technique as that described above. This technique reduces the above description to

```
(complete ()
   (with-wires (c-/1 c-/2)
      (par
         (process (let loop (choice (c-/1+ (loop)) ())) a+
                  (let loop (choice (c+ (loop)) ())) a-)
         (process c+ (choice (c-/1+ :: c-) (c-/2+ :: c-)))
         (process (let loop (choice (c-/2+ (loop)) ())) b+
                  (let loop (choice (c+ (loop)) ())) b-))))
```

This description generates the precise net pictured in figure 37.

This chapter has presented work that builds on previous work in trace structures [15], safe Petri nets [43], and composition of labeled Petri nets by merging transitions [53] in several ways. First, we allowed the definition of synchronization groups to control how synchronization is performed. Second, we considered nets labeled with a set of actions, rather than a single action, and we defined a structural property that allows us to use an interleaving behavioral semantics rather than the more computationally expensive powerset or multiset semantics. Third, we defined a syntactic shorthand for composition and receptiveness that avoids an exponential blowup in the size of the resulting nets. Each of these extensions contributes to our ability to model, specify, and analyze circuit behavior.

Chapter 5

Conformance, Verification, and Simulation

Mirrors should reflect a little before throwing back images. —Jean Cocteau, 1889–1963

We have introduced a formalism for describing circuits and their behavior. In this chapter, we discuss what it means for an implementation to meet a specification and describe how we automatically compute this relation. This chapter is primarily concerned with the implementation of David Dill's verification methods based on trace theory within the Orbits formalism.

In general, simulation determines whether a circuit works for some circumstances and its approximate performance. Verification determines whether a circuit works under all circumstances and the exact bounds on its performance. Both operations are important. Simulation examines sample behaviors of the circuit to make sure that the description of the circuit and its specification are approximately correct. Since many circuits and designs are too complex for formal verification, efficient simulation that considers a large number of possible traces is important.

Verification, on the other hand, provides a simple yes or no answer (and with a no answer, usually some sample behavior that illustrates the failure). If a circuit has been verified as correct, then the designer can be much more certain that the circuit will not fail—provided that the verification system and its formalism, the circuit models, and the specification are sufficiently reliable and accurate. (In practice, when a negative result is obtained, the circuit models and specification are carefully scrutinized for bugs. Any positive result is often considered proof that a circuit performs correctly. Yet, a poor circuit model, incorrect specification, or inappropriate formalism can all yield a false positive result.)

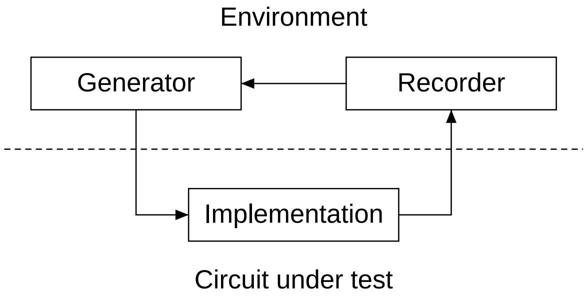


Figure 38. Verification or simulation test bed.

5.1 Safety and Liveness Properties

There are two main types of properties that can be checked of concurrent systems: safety properties and liveness properties. Safety properties stipulate that the system will never enter a bad state. Liveness properties assert that eventually some good state will occur. We only consider safety properties in this dissertation.

This is not as limiting as it might seem. For instance, when designing a cache controller, knowing that we will get a memory word eventually may not be enough; we insist that we get that memory word within one tenth of a microsecond. Many liveness properties can be converted to safety properties by adding a real-time maximum delay before the expected event happens. The next chapter extends our representation and trace theory to handle just such real-time constraints.

5.2 Simulation Uses Verification Techniques

The primary difference between simulation and verification is that verification is, in a sense, exhaustive simulation. By exhaustive, we mean that for the properties of relevance, every state (or, with advanced techniques, some representative of every congruence class of states) is examined. In simulation, we primarily are interested in examining as many of these states or congruence classes as we can.

In both cases, we construct a composite machine like that in figure 38. We model the circuit we are testing. In addition, we have some type of generator that creates inputs for the circuit under test. We also have a recorder that records the outputs or checks them against specific requirements. Finally, the recorder and the generator are connected to reflect how the outputs of the circuits affect the environment, which in turn affect the inputs applied to the circuit. The recorder and generator together make up the circuit environment. The entire system is closed; there are no unconnected inputs.

When comparing a specification with an implementation, all of the information needed about the circuit environment is captured in the specification. Thus, the generator and recorder are derived from the specification, for both simulation and verification.

Simulation and verification do have some differences. Simulation runs might maintain a statistical trace of relevant output events, and the input events might be generated according to some particular distribution. No specific record of visited states is kept, and backtracking information (so that prior states can be restored and a different alternative explored) are not maintained. For verification, a full table of states visited must be kept so that the verification algorithm itself can terminate when all states have been seen.

There are some interesting techniques that fall somewhere between verification and simulation. For instance, with large state spaces, verification can often be performed by maintaining a partial table of states seen; as memory gets tight, various states are removed as new states are encountered. Usually such removal is done in a random fashion. Where some reasonable fraction of the total state space can be kept in memory, fully accurate verification that is often faster (because paging is avoided) than straightforward verification often results [22].

Similarly, simulation results can often be improved by maintaining a table of some commonly seen states, so that less common states can be explored. This is particularly helpful in finding those failures that occur only when all of several events, each of low probability, occur.

5.3 Conformance Relations

To perform verification, we must define what it means for an implementation to meet a specification. Research in concurrency has generated several different types of *conformance relations* that compare the behaviors of concurrent systems. In this section, we briefly discuss several—language inclusion, failures-divergence refinement, bisimulations, and trace structures conformance—to explain our selection of trace structures conformance as the appropriate conformance relation for circuits.

There are many flavors of concurrency algebras and representations, each with its own relation for comparing specifications and implementations. For example, ordinary deterministic automata (and the languages they accept, the class of regular expressions) are generally compared by simple language inclusion. CCS [35] [36] is usually associated with bisimulations. CSP [20] is often associated with failures-divergences refinement. In trace theory [15], trace structures are generally compared by trace structure conformance.

There are subtle differences between these different types of conformance relations. We consider each in turn and discuss how appropriate each is for verification of circuits.

We also identify two important characteristics of circuits that must be reflected in the formalism used to reason about them: the asymmetry of communication between circuit elements, and the necessity for avoiding all failures. These characteristics make the choice of trace structure conformance the only appropriate approach.

5.3.1 Language inclusion

Language inclusion is the simplest possible relation we might consider for conformation. Indeed, language inclusion captures many of the properties we are interested in.

Consider, for instance, a circuit that has no inputs. In this case, if the circuit models cannot fail, and if the set of output traces generated is a subset of the output traces allowed by the specification, then the circuit performs correctly, at least as far as safety constraints are concerned. If the circuit model fails, that means that the model of an internal element was subjected to inputs from another internal component that were illegal. For example, the model of an internal inverter might indicate that if the input changes twice without an output event between them, then the circuit will fail. Thus, any failure of any component of the circuit under test can be considered a global failure; either the circuit has a real flaw, or the model of some element of the circuit needs to consider additional input conditions.

If we consider a circuit with only inputs, then we find the situation quite different. In this case, the circuit must be able to handle all possible combinations of input events given by the specification—possibly more, but certainly no less. Thus, for input events, the specification must generate a subset of the traces that the implementation accepts.

Any formalism used to reason about circuits must distinguish inputs from outputs, and the conformance relation must treat them differently. It is possible to construct a conformance relation based on regular expressions and language inclusion by considering prefix-closed regular trace sets and observing the first symbol in each trace in the difference sets between the implementation and specification and vice-versa. For instance, ignoring failures, if the language (or success set) of the implementation is L^M, and the language of the specification is L^S, then we might define conformance by

$$(L^M - L^S) \subseteq (L^S I - L^S)\Sigma^* \wedge (L^S - L^M) \subseteq (L^M O - L^M)\Sigma^*$$

This says that any additional behaviors allowed by the circuit are caused by an input not allowed by the specification, and any additional behaviors allowed by the specification start with an output.

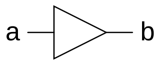
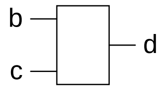
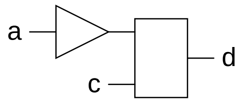
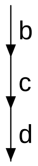
Circuit			
Language	$pref(ab)$	$pref(bcd)$	$pref(acd)$
Synchronization Tree			
Failure Sets	$\epsilon\{b\}$ $a\{a\}$ $ab\{a, b\}$	$\epsilon\{c, d\}$ $b\{b, d\}$ $bc\{b, c\}$ $bcd\{b, c, d\}$	$\epsilon\{c, d\}$ $a\{a, c, d\}$ $ac\{a, c\}$ $acd\{a, c, d\}$
Trace Structure Sets	$S = pref(ab)$ $F = \{aa, aba\}\Sigma^*$	$S = pref(bcd)$ $F = \{c, bb, bcb, bcc, bcdb, bcdc\}\Sigma^*$	$S = pref(acd)$ $F = \{c, aa, ac\}\Sigma^*$

Figure 39. An illustration of silent actions introduced by composition and hiding. The notation $pref(a)$ means the set of all prefixes of a, including a.

As it turns out, the trace conformance we end up using looks a lot like this. But we must consider failure traces as well. This section explains the inclusion of nondeterminism and failures in our model by showing how they are introduced by composition and hiding.

Consider a circuit with two inputs b and c and one output d. The failure-free behavior of the circuit is described by the sequence $[b, c, d]$ and its prefixes. If the input c arrives before the input b, that is a failure; our model does not allow that behavior. Now imagine that we attach a buffer with input a and output b, with behavior $[a, b]$ and its prefixes. The composite behavior (which is approximately identical for all of the formalisms we consider) can be described by the sequence $[a, b, c, d]$. Now, if the environment provides input a, and then provides input c without waiting for b, we have a failure.

Imagine now that we hide wire b. Now the visible behavior can be described by the sequence $[a, \tau, c, d]$ where τ means an internal, invisible action. After the environment provides a, it no longer has any way to tell what state the circuit is in, that is, whether the internal action has occurred or not. If the environment tries to supply a c, it might be refused (if the internal action has not occurred yet) or it might be accepted. Thus, the language of this composite circuit is $[a, c, d]$, yet this sequence might be a success or a failure, depending on an internal choice the circuit makes.

Note how these internal actions are just like the unlabeled transitions that were discussed in the previous chapter. While those were undesired transitions introduced by language-preserving Petri net constructions, these are a natural consequence of composition and hiding.

In most circuits, almost all of the wires are hidden; most events in the circuit are internal. Thus, considering internal actions and determining whether a particular visible event trace is a success, failure, both, or neither for a particular circuit are of primary concern. Therefore, simple language inclusion semantics fails to capture our intuitive notion of circuit conformance.

Essentially, failures represent sequences that might not be accepted by a nondeterministic finite state machine. CCS, CSP, and trace theory each have interpretations of the behavior of such nondeterministic state machines. We will consider each in turn.

5.3.2 Bisimulations

CCS [35] [36] introduced *synchronization trees* that can illustrate many subtle nondeterminism and synchronization examples. A synchronization tree is a state machine where each state has only a single edge into it and the arcs are labeled by actions or τ, the silent action. Each state can have multiple outgoing arcs with the same action label. The synchronization trees for our example are given in figure 39.

The primary equivalence relation used in CCS is the bisimulation. In general, the operands in CCS are agents, which can be represented by a nondeterministic state machine and a set of states. (Bisimulations have also been applied to state machines, Petri nets, and other formalisms.) A bisimulation S is a relation between agents, such that for all a in W, if a pair (P, Q) is in the bisimulation, then:

$$\text{If } P \xrightarrow{a} P', \text{ then } Q \xrightarrow{f(a)} Q', \text{ and } (P', Q') \in S.$$
$$\text{If } Q \xrightarrow{a} Q', \text{ then } P \xrightarrow{f(a)} P', \text{ and } (P', Q') \in S.$$

Here, $P \xrightarrow{a} P'$ means that P after execution of a results in P'. There are different types of bisimulations depending on what the set W and the function f are. For instance, a might be chosen from Σ, or Σ^*, or $\Sigma \cup \{\tau\}$. Typically f is the identity function, but it might be a function that removes τ actions from a sequence of actions.

Bisimulations can capture an amazing degree of behavioral equivalence. For instance, one strong equivalence relation on Petri nets is a bisimulation where W is the set of partially-ordered multisets on A; this bisimulation even distinguishes the behavior of the rightmost net in figure 32 and the net in figure 33 in the previous chapter.

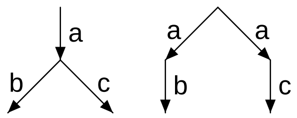


Figure 40. Automata that cannot be distinguished by language equivalence but can be distinguished by failures equivalence.

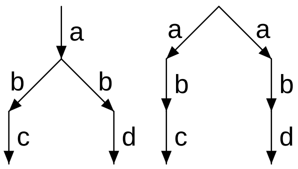


Figure 41. Automata that cannot be distinguished by failures equivalence but can be distinguished by bisimulations.

Bisimulations are currently popular because they are a behavioral relation that can fully reflect the branching structure of internal choices. For our purposes, however, bisimulations are too strong; we want a conformance relation, not an equivalence relation. Also, if the circuit might make a nondeterministic choice that will lead to a failure for a particular trace of events, we must consider (conservatively) that it will indeed make that choice; we need not consider when it makes the choice.

5.4 Failures Refinement

CSP [20] uses a behavioral trace semantics that more closely fits what we require. We only consider failure semantics here; failure-divergence semantics also considers liveness properties, which we are not concerned with. The (non-divergent) behavior is characterized by a failure set. Each element of the failure set is a sequence of possible actions, followed by a set of sets of possibly refused actions (the refusal set). The failure sets are also given in figure 39. Often the refusal set for a particular possible action is flattened by taking the union of all of the sets of actions that might be refused; this is called the maximal refusal set.

Failure set equivalence falls between simple language equivalence and bisimulation equivalence. Language equivalence cannot distinguish the automata in figure 40, but these automata can be distinguished by failures equivalence. On the other hand, failures equivalence cannot distinguish the automata in figure 41, which can be distinguished by bisimulations.

In addition, CSP has a refinement relation; if the failure set of one automaton is a subset of the failure set of another, then the second refines the first; this is the style of conformance relation we seek.

Unfortunately, failures refinement does not allow an implementation to exceed a specification; that is, it does not allow an implementation to accept a larger set of inputs than the specification provides for. It is probably possible to add this feature to failures refinement by extending the specification in the following way: for any trace a that is possible, and input action b, such that ab is not in the set of possible traces, extend the trace set with $ab\Sigma^*$ and a refusal set of Σ for each such added trace.

But CSP is both too rich and too poor for what we need; CSP contains different types of choice reflecting the difference between internal choice and environmental choice. In our case, the choice depends on whoever generates the action; in a choice between two input actions, the environment makes the choice; in a choice between two output actions, the circuit makes the choice; in a choice between an input and an output action, a race condition holds in which the choice is made by whichever occurs first. In addition, we need receptiveness; while it can be added to CSP, it has not yet been in any existing tools that we are aware of.

5.4.1 Circuit characteristics

Circuits have three primary characteristic properties that determine our choice of semantics. The first is the asymmetry of communication in circuits; one element generates an event, and others observe the event; the elements that are observing cannot prevent that event from occurring. The second is that, where nondeterminism exists, we must assume that the worst possible situation will occur. The third is that an implementation can exceed a specification and still conform; we require a relation that reflects this property.

Asymmetric communication is more common in the real world than the symmetric communication that is used by CCS and CSP. (CSP has an asymmetry, but only in the sense of what is sending and what is receiving a particular data item; the blocking semantics are the same for both ends.) In almost any communication between agents, one agent initiates the communication, and the other responds. A person mails a letter or dials the telephone; the recipient cannot (normally) prevent the letter from appearing in the mailbox or the telephone from ringing. Most forms of communication that can be considered symmetric (a conversation, for instance) are alternating asymmetric communications (each speaker generally takes a turn). While there exist fully symmetric communications (a car crash is such an example [24]), these are less common than asymmetric communications.

For directed communication, the actions in our alphabets are separated into two classes, input actions and output actions. We require that each element be receptive on the input actions. Conformance is defined in terms of the allowed set of output behaviors for a given set of input behaviors, ignoring those input behaviors that are not allowed by a specification.

The second characteristic of a circuit is that, if it can choose to fail, we must assume that it will fail. A circuit that might fail is not of much more use than a circuit that will definitely fail. If a specification includes a failure trace, then that particular trace can always be considered a failure and can be excluded from the set of traces that the implementation must be tested on. Conversely, if an implementation can fail on a trace, then it must be considered to always fail on that trace.

The trace structure theory as extended by Dill has all of the properties we require. While trace theory has much in common with CSP trace semantics, it has been developed especially for reasoning about circuits and thus includes separate input and output actions, receptiveness, and failure exclusion.

5.5 Trace Theory

In this section, we give a brief overview of trace theory. A much more formal and complete exposition is given in Dill's dissertation [15].

Intuitively, a circuit is failure-free relative to a specification when the circuit allows all inputs that the specification allows and generates a set of outputs allowed by the specification. The circuit also must not fail internally; no component of the circuit can fail given a set of inputs allowed by the specification. More precisely, in any environment in which the specification does not fail, an implementation must also not fail. A trace structure with an empty failure set cannot fail and is characterized as failure-free; a trace structure with a non-empty failure set can fail and is not failure-free. Trace theory shows that any context or environment can itself be reduced to a trace structure. Thus, given an implementation TS^M and a specification TS^S, the implementation conforms to the specification (written $TS^M \preceq TS^S$) if and only if for every TS (representing a context or environment), if $TS^S \parallel TS$ is failure-free, so is $TS^M \parallel TS$. The relation $\preceq$ has certain properties, including monotonicity with respect to composition and hiding, reflexivity, and transitivity. Transitivity is particularly useful because it allows hierarchical verification.

Dill defines conformation equivalence as $TS \sim_C TS'$ if and only if $TS \preceq TS'$ and $TS' \preceq TS$. The primary purpose of conformation equivalence is in justifying certain simplifications that can be performed on trace structures. Specifically, he defines a procedure for finding a canonical form for an equivalence class of trace structures that are all conformation equivalent. This canonical form is important, so we will discuss it in more detail here.

To calculate the canonical form for a given trace structure, two specific operations must be performed on it: autofailure manifestation and failure exclusion.

Autofailure manifestation is based on the observation that, if the last action a circuit performs before a failure is an output, then the circuit causes its own failure; the environment cannot prevent the circuit from failing. Thus, the prefix of the trace created by deleting that output event is also considered a failure. Put another way, any failure trace that ends in an output must also have its immediate prefix in the failure set. We replace the failure set with $(F/O^*)\Sigma^*$, where F/O^* denotes the set of traces f such that there exists some $o \in O^*$ such that $fo \in F$.

Unless the success set is empty, every failure has some prefix that is a success. After autofailure manifestation, the action on which the circuit fails is thus always an input.

The second transformation is elimination of successes that are also failures from the success set; if the circuit might fail, it might as well always fail. Thus, we replace the success set with $S' = S - F$.

If these two operations are performed in order, we end up with a canonical trace structure with a disjoint success and failure set, and with the failure set exactly equal to $((SI \cup \{\epsilon\}) - S)\Sigma^*$. Thus, a canonical trace structure can be fully characterized by its set of input and output actions and its success set. This is a strong and useful result.

If the original success and failure sets are regular, so is the canonical success set. Thus, we can construct a deterministic finite state generator. This turns out to be straightforward; both autofailure manifestation and failure exclusion are readily combined with the conversion of a nondeterministic finite state machine to a deterministic one, and the total operation yields a finite state machine that generates the success set of the canonical trace structure. Having eliminated the nondeterminism, we can check for conformance using the simple language inclusion expression given earlier in the chapter by exploring the product machine of the specification and the implementation.

The canonicalization procedure can be expensive, especially when considering the implementation, which might be composed of many concurrent machines. When calculating the canonical form, all reachable states of the implementation are considered, some many times. In checking for conformance, however, we are only interested in those trace sequences which are successes for the specification, and any failure is cause for immediately giving a negative result. Therefore, we would like to find a faster technique that does not require finding the canonical form of both the specification and the implementation. The trace structure *mirror* operation gives us such a technique.

5.6 Mirrors

One way to test conformance is to construct each possible environment, determine whether it is failure-free when composed with the specification, and if so, determine whether it is also failure-free with the implementation. Since there are so many trace structures, we must reduce the number.

First, if $TS' \preceq TS$, and TS is failure-free with both the environment and the implementation, then we do not need to test TS'. We would like to find some trace structure TS that is failure-free when composed with the specification, and to which every other trace structure TS' that is failure-free when composed with the specification conforms. This trace structure depends only on the specification. If we can find such a trace structure, then our conformance check will be reduced to checking if the composition of TS and the implementation is failure-free. Thus, such a trace structure is our entire test bed for checking any implementation against the specification.

Dill shows that for any trace structure TS, there is a unique maximum environment that meets our requirement. This environment is called the mirror of the specification. If the specification is in canonical trace structure form, then a canonical trace structure for the mirror can be formed by switching the input and output alphabets, leaving the success set alone, and calculating a new failure set according to the same formula we use for any canonical trace structure. Thus, the deterministic finite state acceptor for the specification is precisely the same as that of the mirror.

Because of this, we do not find the canonical form of the implementation in order to calculate conformance. Instead, we find the canonical form of the specification, swap its inputs and outputs, ensure that it is receptive on its new inputs, compose it with the implementation, and determine if the resultant trace structure is failure-free. We do this by exploring the state space of the component automata.

We still must find the canonical form of the specification. Normally the specification is quite a bit simpler than the implementation, so this is generally inexpensive compared to calculating the canonical form of the implementation. In addition, the specification restricts the set of traces that are explored in the implementation, so we explore only a subset of the possible states of the implementation. This provides a powerful technique for verification; we explore only the useful state space of the implementation.

This same technique is used in verification tools other than Dill's verification system `AVER`. For instance, FDR, a failures-divergence checker for CSP, uses an almost identical technique. The specification is normalized—that is, reduced to a deterministic state machine with maximal refusal sets for each state—and then the state space of the implementation is explored in conjunction with the specification. In all of the real-world examples they have examined, the normalization of the specification is fast and improves the performance of the tool.

We provide three primitives that can be used to compute conformance.

(`mirror` *circ*) Given a circuit, returns another circuit that has the input wires and output wires switched. This does not calculate the canonical form for a circuit, nor does it reduce or in any other way check the circuit.

(`normalize` *circ* `.` *args*) Given a circuit, returns another circuit that is a deterministic state machine with autofailure manifestation and failures exclusion applied. This explores the entire state space of *circ*, saving all edges, so it might be expensive. The resultant circuit preserves wire initializations of the original. The circuit must not have any timing associated with it. If *circ* has arguments other than wires, they must be known and supplied in *args*. Initial values must be given to all input wires within *circ*.

(`verify` *exp*) Given an expression, it builds the orbital net and computes the reachable state space in depth-first fashion. If a failure state is reachable, it displays the failure trace and then executes a breadth-first search to find another, usually shorter, failure trace. The result is undefined.

So, in general, given a specification and an implementation as circuits, and assuming that the implementation is already receptive on its inputs, we test for conformance with the sequence

```
(def normal-spec (normalize specification))
(def mirrored-spec (mirror normal-spec))
(def rec-mir-spec (receptive mirrored-spec))
(verify
   (with-wires global (... input and output wires ...)
      (par
         (rec-mir-spec ... output wires ... input wires ...)
         (implementation ... output wires ... input wires ...))))
```

As it turns out, most specifications are already normalized by default. When we introduce timing and data flow, it will turn out that normalization is quite a bit more complex; so for specifications with these features, we will require that the input specification already be normalized.

Indeed, normalization, at least to a simple deterministic state machine, has disadvantages. First, it eliminates all possible concurrency from a specification. It is possible to envision a normalization procedure that retains concurrency, but we leave this for future research. Concurrency is helpful for many algorithms; for instance, our timing algorithm exploits concurrency to calculate the reachable state space more quickly. Various state space reduction techniques take advantage of concurrent actions by only exploring one possible interleaving. Also, the normalized form may have many more places and transitions (in orbital net form) than the original net.

The concept of normalization does not require that behavior be reduced to a minimally concurrent state machine, but only some representation that generates all the traces of that state machine. Future versions of Orbits might well yield a more concurrent normalized form; the limitations of normalization are strictly a function of the current implementation.

Even with these drawbacks, normalization is often useful. For instance, it allows hierarchical verification without requiring intermediate-level specifications; common idioms such as cross-coupled nor-gates can be normalized, and the normalized specification used instead of two instantiations of the nor model. This is particularly useful for delay-insensitive circuits, where the normalized form of a series of gates is often much smaller than the original specification.

5.7 Buffer Examples

As a simple example to help motivate trace structures, let us consider once again modeling a buffer with delay. A basic model is:

```
(def (buffer (a) (b))
   (process a+ b+ a- b-))
```

If we add receptiveness to this model, then any input change while an output change is pending will be a failure.

```
(def rec-buffer (receptive buffer))
```

Now we explore the behavior of two concatenated buffers.

```
(def (buffer-imp (a) (b))
   (combinational (rec-buffer (rec-buffer a)) b))
```

Rather than creating the intermediate wire with `make-wire`, we use the macro `combinational` to automatically construct the circuit from a simple expression. This only works when each output feeds only a single gate.

`(combinational` *expr out-wire*`)` This macro constructs a circuit from a simple expression, allocating all necessary intermediate wires, and connects the output of the final element to *out-wire*. Each element of the list must have just a single output. The recursion works as follows: If the *expr* is a number or a wire, this value is returned directly. If *expr* is a list, then its first element must be the name of a circuit. The arguments are evaluated according to this procedure recursively, passing a separate new temporary wire as the output wire for each recursive invocation. Then, the circuit macro is called, using the set of returned values, along with the current output wire inserted in the appropriate place in the list. Finally, the current output wire is returned as the value. As an example,

```
(combinational (fast-and (fast-not a) (buffer b 12 13)) c)
```

is equivalent to

```
(with-wires (t@1 t@2)
   (par
      (fast-not a t@1)
      (buffer b t@2 12 13)
      (fast-and t@1 t@2 c)))
```

Do these composed buffers conform to a single buffer?

```
(def buffer-mirror (mirror buffer))
(def rec-buffer-mirror (receptive buffer-mirror))
(verify
   (with-wires global ((a 0) (c 0))
      (par
         (buffer-imp a c)
         (rec-buffer-mirror c a))))
```

The result of this command indicates that it does, and six states were explored to determine this.

Are the two conformation equivalent? To answer this question, we have to check that the specification conforms to the implementation. In order to do this, we have to normalize the implementation.

```
(def buffer-imp-norm (normalize buffer-imp))
(def buffer-imp-norm-mirror (mirror buffer-imp-norm))
(def rec-buffer-imp-norm-mirror (receptive buffer-imp-norm-mirror))
(verify
   (with-wires global ((a 0) (c 0))
      (par
         (rec-buffer a c)
         (rec-buffer-imp-norm-mirror c a))))
```

This succeeds, examining four states. This tells us that two concatenated buffers are conformation equivalent to a single buffer. As a matter of fact, the net resulting from normalizing a pair of concatenated buffers is precisely the same as the original buffer specification.

5.8 Gate Examples

Now we consider the behavior of primitive gate elements. In particular, we investigate whether the delay aspects of a gate should be modeled separately from or as an integral part of the gate model itself. We find that separating the delay from the function in a gate model yields more flexibility and smaller nets than when considering them together.

Two primary design methodologies used in asynchronous circuit design are speed-independent design and delay-insensitive design. In both of these design styles, the speed of gates is assumed to be arbitrary; a circuit works correctly regardless of the speed of any gate. In a delay-insensitive design, the wires are also allowed to have any arbitrary delay. A speed-independent design assumes that whenever a gate changes, that output change occurs at all of the gates it is connected to at the same time. This assumption is called the isochronic fork assumption. As it turns out, the set of circuits that can be designed in a delay-insensitive manner is much smaller than the set of circuits that can be designed using a speed-independent approach, so the latter is more popular.

In a speed-independent design, a combinational gate can be modeled by a fast combinational element with a delay buffer connected to the output, because all of the delay is associated with output changes. For a delay-insensitive design, the wires are allowed to have arbitrary delay, so a model must also have delays connected to all inputs of a combinational gate.

Now let us examine what happens when we connect `buffer` to the output of a fast and-gate and normalize the result. For simplicity, we will initialize the wire values to zero.

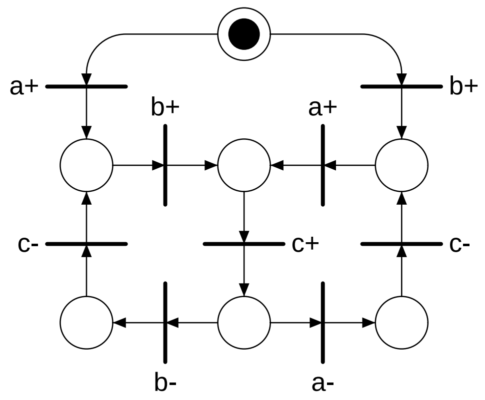


Figure 42. Normalized delay-insensitive and-gate.

```
(def (delay-and (a b) (c))
   (wire-init-set! (a 0) (b 0) (c 0))
   (combinational (rec-buffer (fast-and a b)) c))
(def normalized-and (normalize delay-and))
```

Not surprisingly, this results in exactly the model for the and-gate with delay that Dill uses in his dissertation. The original unnormalized model is still preferable, however, because it has more concurrency and has a smaller net model.

Now we normalize a delay-insensitive model of the and-gate. The default version of `fast-and` that is supplied requires the initial values of all wires to be set. Unfortunately, `combinational` does not give us access to the internal wires so that we can set their values. We slightly revise our `buffer` model to take care of this, and then connect the circuit together and normalize.

```
(def (buffer (a) (b))
   (wire-init-set! (a 0) (b 0))
   (process a+ b+ a- b-))
(def (delay-and (a b) (c))
   (wire-init-set! (a 0) (b 0) (c 0))
   (combinational (rec-buffer (fast-and (rec-buffer a) (rec-buffer b))) c))
(def normalized-and (normalize delay-and))
```

The unnormalized circuit has sixty-eight total states; this is reduced to seven final normalized states. The result is shown in figure 42. Note how, after the initial state, there is no reachable state with both inputs low. In general, for an n-input and-gate in a delay-insensitive circuit, at most one input will be low at a time. For an n-input or-gate, at most one input will be high at a time. Thus, if in some reachable state of a circuit, some and-gate

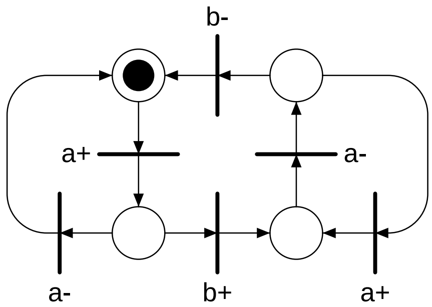


Figure 43. Hazard-ignoring delay element.

has more than one input low or some or-gate has more than one input high, the circuit is not delay-insensitive.

Now let us consider a buffer where a hazard is not necessarily fatal; the buffer ignores hazards, with no output change. The state machine for such a buffer is shown in figure 43. We describe such a buffer with

```
(def (buffer (a) (b))
   (wire-init-set! (a 0) (b 0))
   (complete ()
      (sm loop ((s 0))
         (guard
            ((= s 0) a+ (loop 1))
            ((= s 1) a- (loop 0))
            ((= s 1) b+ (loop 2))
            ((= s 2) a- (loop 3))
            ((= s 3) a+ (loop 2))
            ((= s 3) b- (loop 0))))))
```

This buffer conforms to the original buffer specification, but the original one does not conform to this one. In addition, two of these buffers conform to the original buffer specification, but two of these buffers do not conform to the new specification. The failure trace is $[a{+}, a{-}, c{+}]$—the intermediate wire can remember the hazard and cause an output after both events of the hazard.

When we explore composing our fast and-gate with this new style of buffer, we find the result is an and-gate model that is behaviorally equivalent to the first and-gate model we considered in chapter two. That model does have some advantages—its size is linear in the number of inputs, while the other models we considered are exponential in the number of inputs.

These examples make it clear that separating the combinational function from the delay characteristics of a gate yields a great deal of flexibility. When we introduce real time, we will find that this ability becomes even more important.

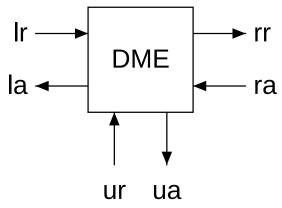


Figure 44. A distributed mutual exclusion cell.

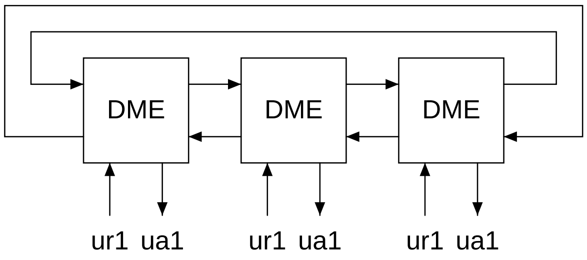


Figure 45. A ring of three DME cells, only one of which has the token at any given time.

5.9 Circuit Examples

We now present one final, more complex example taken from [15]. The circuit is a distributed mutual exclusion element cell, pictured in figure 44. For n-way mutual exclusion, n of these circuits can be connected in a ring, as shown in figure 45.

The circuit has three interfaces: a left interface, a right interface, and a user interface. The abstract model is that of token transfer; only the mutual exclusion cell with the token can pass it to its user, and only one mutual exclusion cell has the token at a time. The left and right interfaces are used to transfer the token to and from adjoining mutual exclusion cells; the user interface is used to transfer the token to and from the user. Each interface obeys a four-phase protocol, with a positive transition on the request wire indicating a request, and a positive transition on the acknowledge wire indicating the transfer of the token. The user returns the token to the mutual exclusion cell by a negative transition on his request line; all other negative transitions are strictly resets and do not indicate the transfer of the token. Finally, we wish the mutual exclusion cell to be lazy; if it does not have the token, it will not try to get the token until either its user or its left interface requests the token.

We specify the operation of the circuit by dividing it up into specifications of the three interface protocols and specification of the token transfer.

```
(def (dme (lr ur ra) (la ua rr) tok)
   (wire-init-set! (lr 0) (ur 0) (ra 0) (la 0) (ua 0) (rr 0))
   (complete (par
      (process lr+ la+ lr- la-)
      (process ur+ ua+ ur- ua-)
      (process rr+ ra+ rr- ra-)
      (sm loop ((s tok) (reqs 0))
         (guard
            ((< reqs 2) lr+ (loop s (+ reqs 1)))
            ((< reqs 2) ur+ (loop s (+ reqs 1)))
            ((and (= s 1) (> reqs 0)) la+ (loop 0 (- reqs 1)))
            ((and (= s 1) (> reqs 0)) ua+ (loop 2 (- reqs 1)))
            ((and (= s 0) (> reqs 0)) rr+ (loop s reqs))
            ((and (= s 0) (> reqs 0)) ra+ (loop 1 reqs))
            ((= s 2) ur- (loop 1 reqs))))))
```

The state machine that handles token transfer maintains two items of state: the number of outstanding requests for the token, and what component has the token. If `s` is 0, some other DME cell has the token. If `s` is 1, we have the token. If `s` is 2, then our user has the token. The initial value of the token is passed in as an extra argument `tok`.

The first two lines of the state machine increment the number of requests as we receive them; we must have the Boolean condition `(< reqs 2)`; otherwise the enumeration of states will not terminate. The second two lines represent transfer of the token to the left cell and to the user, respectively. The fifth line indicates a request for the token from the right interface, and the sixth line represents the acknowledgment of that request. Finally, the last line represents transfer of the token back from the user.

By separating the specification into four pieces, each concerned with a particular aspect of the behavior, we have a more concurrent, shorter, and more easily understood description. The composition we use here is not the composition of circuit algebras and receptive trace structures, but rather the more traditional composition of concurrent automata. The former is used to model the composite behavior of two circuit components; the latter is used to describe complex behavior out of simpler behaviors.

We mirror the specification of the distributed mutual exclusion cell and make it receptive:

```
(def dme-mirror (mirror dme))
(def dme-spec (receptive dme-mirror))
```

Martin's implementation of the distributed mutual exclusion element [30] is given in figure 46. All of the gates in the circuit are provided as fast delay-free elements, except for the S/R flip-flop, the ME element, the C element, and the half-and-gate. The C element is

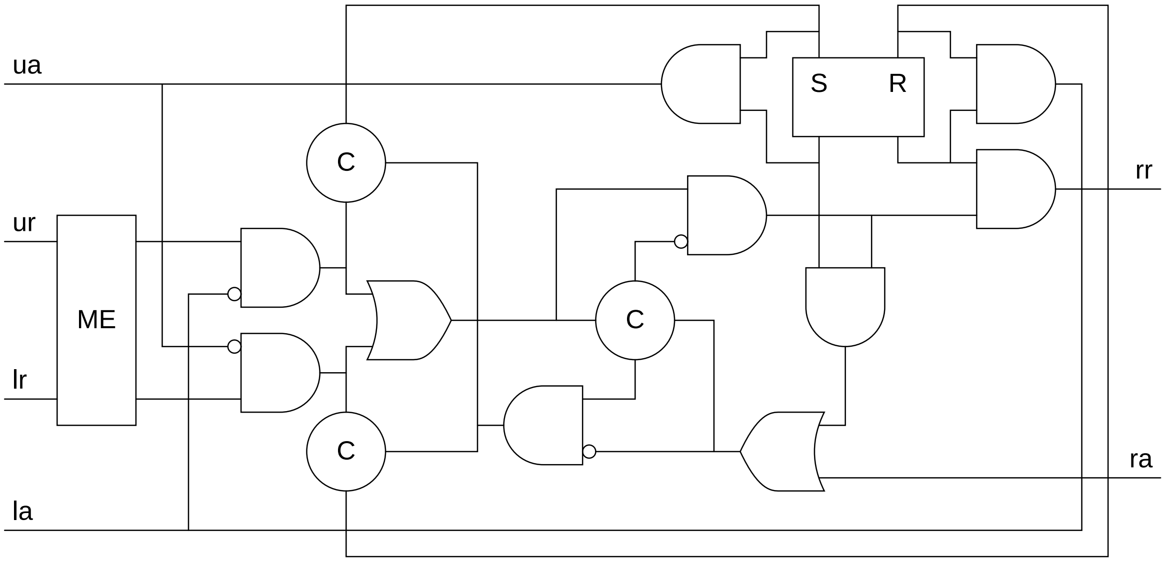


Figure 46. Martin's distributed mutual exclusion circuit.

provided as an atomic element with delay. We define the half-and with `sm-ckt`. We use `si` to add the appropriate delays to the existing fast elements.

(`si` *circ*) Adds the buffer defined as `buffer` to all outputs of the circuit and returns the resulting circuit. The `si` primitive is typically used for modeling gates in a speed-independent manner, but if `buffer` is defined differently, it can also add any particular form of delay.

(`di` *circ*) Adds the buffer defined as `buffer` to all of the inputs of the circuit, and returns the resulting circuit. The `di` primitive is typically used for delay-insensitive design.

```
(sm-ckt (fast-halfand (a b) (c)) (and a (not b)))
(def /nor/ (si fast-nor))
(def /and/ (si fast-and))
(def /or/ (si fast-or))
(def /halfand/ (si fast-halfand))
```

We define a set-reset flip-flop with two cross-coupled nor-gates. We pass in the initial state of the flip-flop as an additional element:

```
(def (/sr/ (s r) (q notq) init)
   (wire-init-set! (q init) (notq (not init)))
   (par
      (/si-nor/ notq r q)
      (/si-nor/ q s notq)))
```

A mutual exclusion element is easy to define. We enforce the two-phase protocol on both sets of request and acknowledge wires and also enforce mutual exclusion on the token. We make the result receptive:

```
(def (me (i1 i2) (o1 o2))
   (wire-init-set! (i1 0) (i2 0) (o1 0) (o2 0))
   (process i1+ o1+ i1- o1-)
   (process i2+ o2+ i2- o2-)
   (process (choice (o1+ o1-) (o2+ o2-)))
(def /me/ (receptive me))
```

Finally, we put the circuit together. We introduce all necessary wires. Note how we only supply the initial value of a few of them, as the rest are calculated by the gate models:

```
(def (/dme/ (lr ur ra) (la ua rr) tok)
   (wire-init-set! (ur 0) (lr 0) (ra 0) (ua 0) (la 0) (rr 0))
   (with-wires (me1 me2 a1 a2 (a3 0) a4 a6 o1 (o2 0)
                c1 c2 c3 (q tok) (nq (not tok)))
      (par
         (/me/ ur lr me1 me2)
         (/halfand/ me1 la a1) (/halfand/ me2 ua a2)
         (/c/ a1 a3 c1) (/c/ a2 a3 c2) (/c/ o1 o2 c3)
         (/or/ a1 a2 o1) (/or/ ra a6 o2)
         (/and/ nq c2 la) (/and/ c1 q ua)
         (/and/ a4 q a6) (/and/ nq a4 rr)
         (/sr/ c1 c2 q nq tok)
         (/halfand/ o1 c3 a4) (/halfand/ c3 o2 a3))))
```

Verification is done as always:

```
(verify
   (with-wires global (ur lr ra ua la rr)
      (par
         (/dme/ lr ur ra la ua rr 1)
         (dme-spec la ua rr lr ur ra 1))))
```

And we find that the circuit works, as expected. We examine 160 states in about a quarter of a second to verify the circuit, using the buffer model where any hazard is a failure. If we instead use the buffer model that can nondeterministically ignore hazards, we have the same number of states and the same performance; we expect this because the hazard-ignoring buffer conforms to the buffer that flags all hazards as errors.

We now consider some experiments on this circuit that were not performed in [15]. If we examine the circuit elements we used, almost all of them are standard library elements except for the half-and gate. The reason that the half-and gate is not a common library element is because most library elements invert all inputs. If all inputs are inverted, usually a gate of the opposite polarity is used, and the output inverted; this matches the speed-independent design model because all additional delay is added to the output. If one input is inverted and another is not, the gate is normally constructed from more than one primitive gate. For instance, the half-and gate can be constructed either by a nand-gate with one input and the output inverted, or by a nor-gate with the other input inverted. The input with the additional delay will cause a violation of the isochronic fork assumption.

Let us consider which implementations of the half-and can be successfully used in the distributed mutual exclusion circuit. We first try the nor-gate with one input inverted:

```
(def /not/ (si fast-not))
(def (/halfand/ (a b) (c))
   (combinational (/nor/ (/not/ a) b) c))
```

Verification finds that a hazard occurs in one of the half-and-gates connected to the middle C element. Depth-first search returns a trace of length twenty-six; breadth-first search returns a trace of length twenty-four. Interestingly, if we use the hazard-ignoring buffer model rather than the hazard-failure buffer model, then the trace reported by depth-first search is 226 actions long, and the trace reported by breadth-first search is twenty-seven actions long. In this and other examples that have been tried, failure traces returned with the hazard-ignoring buffer model tend to be much longer than trace returned with the hazard-failure model; the effects of a hazard have to propagate to the outputs with the former model before an error is noticed, while the latter model flags any hazard immediately.

We next try the alternative version of half-and:

```
(def (/halfand/ (a b) (c))
   (combinational (/and/ a (/not/ b)) c))
```

With this definition, a different hazard is found in the other half-and gate connected to the middle C element. Experimenting with mixtures of the two types of half-and-gates shows that only one combination—using three of the first type and one of the second type of half-and-gates—functions correctly in the speed-independent model.

5.10 State Explosion

The primary problem with finite-state verification is state explosion—a complex implementation might have too many states to explore with the available computational power. Thus, we need a set of techniques to attack the state explosion problem.

We divide the techniques into two categories: automatic and manual techniques. One automatic technique is the stubborn set methods of Valmari [50]. This algorithm creates a reduced reachability graph that reflects all of the properties being checked. The search space is primarily reduced by eliminating sets of interleavings of concurrent transitions that can be shown to have the same effect as a single representative interleaving. Unfortunately, stubborn set approaches rely on finding sets of transitions that are set-wise persistent; enabled transitions in the set cannot be disabled by transitions outside the set, and disabled transitions in the set cannot be enabled by transitions outside the set. If such a set can be found, then only transitions in this set need be considered in calculating the next state. This approach does not appear to work well for circuits because of the high degree of conflict and feedback in circuit models.

Other automatic techniques, such as the unfoldings and configurations technique of McMillan [31], show more promise. Indeed, with the orbital net models and the verification procedure we have, such methods are a natural extension of this research.

Manual techniques for state space reduction can have tremendous impact. For instance, for circuits where the polarity of an event (whether it is a rising edge or a falling edge) is independent of the semantics of that event, it is often useful to use reduced models of gates that are also not concerned with the polarity. This can reduce the state space significantly, but only if every element connected to a particular wire uses the simpler semantics. This technique has the potential of reducing the search space by a factor of two for each such wire found to use two-phase signaling. For instance, we will use this technique in chapter seven to reduce the search space for the Seitz queue element.

Hierarchical verification is also a useful technique, and the `normalize` function can be used for this. With this approach, rather than verify the entire state space of a large implementation, we verify a portion of the implementation separately against some intermediate-level specification, and then use the intermediate specification as the implementation when doing the higher-level verification. Unfortunately, automatic attempts to partition the implementation usually fail, because the state space of the composite circuit given a particular specification is often much smaller than the unrestricted state space of various components of the implementation.

Finally, the use of domain-specific knowledge can have a tremendous impact. Beerel and Burch [5] have found a method that, for certain circuits, can dramatically reduce the cost of verification. Their technique involves finding all memory elements and feedback loops in the circuit and cutting the circuit at these points. An extended specification is constructed that includes all events on the cut points. The combinational elements resulting from the cut are evaluated for hazards using simple Boolean logic against the states reachable in the extended specification. This technique yields impressive results.

A way around the state explosion problem is to use simulation. Simulation will often find failure traces much faster than verification. The primitives supplied in Orbits for simulation are `simulate` and `step`.

`(simulate` *exp*`)` Given an expression, it builds the Petri net and prepares to simulate the net.

`(step` *num-steps*`)` Executes `num-steps` additional transition firings of the most recent simulation (if any).

Chapter 6

Timing

Observe due measure, for right timing is in all things the most important factor. —Hesiod, c. 700 B.C.

Timing is an essential component of circuit design. Data bus specifications, integrated circuit data sheets, and interface descriptions are filled with timing diagrams and timing parameters, sometimes over a hundred different timing parameters for a single integrated circuit. Timing considerations even affect speed-independent design. The isochronic fork and atomic gate assumptions must be checked once the circuit is laid out and delay parameters can be estimated. Also, many speed-independent designs can be simplified when timing is taken into account. The limitations of delay-insensitive circuit design—which allows all gate and wire delays to take on arbitrary values—illustrate that timing considerations cannot be eliminated from circuit design without severe restrictions on the types of circuits that can be designed.

In this chapter, we describe how timing forms an integral part of our representation of circuits. We show how our representation of timing largely preserves the pleasing symmetry between a circuit implementation and its specification, preserving the structural constructions for trace theory operations and allowing hierarchical verification. In particular, we show that two types of timing are required—one type determines the behavior of the net, and the other determines whether a given behavior is a success or a failure. We discuss the difference between discrete and continuous time models and prove that discrete time gives the same results as continuous time for every possible model and specification in our representation. We also describe the concept of conservative and liberal timing approximations and show how that can reduce the state space for verification.

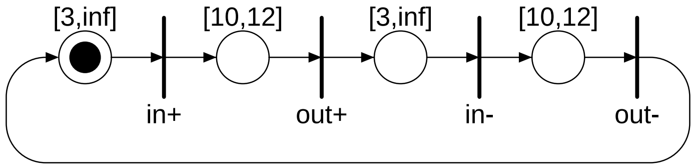


Figure 47. A bounded-delay buffer with timing restrictions on the allowed inputs as well as behavioral restrictions on the possible outputs.

6.1 Adding Time to Orbital Nets

Petri net transitions represent actions, and places represent local state. Since transition firings are instantaneous, it is natural to associate timing with the places rather than the transitions. Therefore, we annotate places with timing information.

There are many different types of timing ranges that can be used. For example, a single delay value is useful for straightforward simulation and analysis. Probabilistic delay distributions can provide accurate simulation results for performance evaluation, but can complicate formal verification. Timing in circuit elements and protocols is normally specified with a minimum and a maximum delay. In Orbits, the timing value associated with a place is a range of values. The minimum value must be a nonnegative number; the maximum value must be a number greater than or equal to the minimum value, or the special value ∞.

Timing values represent both the set of possible output behaviors and constraints on allowed input behavior. Let us consider carefully the difference between these two types of timing information by annotating our simple buffer example with some timing values, as in figure 47. The delay from an output event on wire *out* to an input event on wire *in* is given as $[3, \infty]$; this means that the environment must delay the input to fall within this range. This is a choice made by the environment, and the $[3, \infty]$ indicates that the circuit will perform correctly if the environment chooses these values. Because the buffer cannot affect the delays chosen by the environment, this constraint is a check on the value chosen by the environment. Such delays do not affect the firing sequences that are in the language of the net, but they do distinguish successes from failures; informally, any firing sequence that leaves a token in a place marked with such a timing constraint for a duration outside the range allowed by that timing constraint is a failure.

The $[10, 12]$ timing specification indicates a delay from input events to output events, and this is a choice of the buffer circuit. The buffer can choose any values in this range; in fact, the buffer must be able to choose any such value listed and generate its output event without interference from any other circuit or subnet. Thus, this form of timing constraint restricts the timed language that can be generated by the net.

Let us consider whether this timed buffer conforms to the untimed buffer, or whether the untimed buffer conforms to the timed buffer. We consider a place without timing information to be an untimed place; for the moment, we can consider these as timing places annotated with $[0, \infty]$, indicating that any nonnegative delay is possible between events. (We allow zero here so that events that are not concurrent in a model can still occur at the same time value, and thus the same instant of time, even though they are not simultaneous in the firing sequence.) We assume that there is some global clock variable t that starts at zero and never decreases; at time $t = 0$, all of the tokens in the initial marking appear in their respective places. We introduce timed firing sequences, which are untimed firing sequences with each element associated with a single nondecreasing time value.

Consider the firing sequence $[(in+, 3), (out+, 14), (in-, 15)]$. This firing sequence is in the timed language of the untimed buffer. It is also in the timed language of the timed buffer, but it is a failure since the environmental timing constraint is violated. Thus, since this timed buffer does not accept all of the input sequences allowed by the untimed buffer, it does not conform to the untimed buffer. If the timing constraints from the output events to the input events are removed, then the timed buffer does indeed conform to the untimed buffer.

Now consider the sequence $[(in+, 5), (out+, 6)]$. This firing sequence is in the timed language of the untimed buffer, but it is not in the timed language of the timed buffer. Here, it is an output event that causes the violation; the untimed buffer generates a sequence that is not allowed by the timed buffer. Thus, the untimed buffer does not conform to the timed buffer. If the timing constraints from the input events to the output events are removed from the timed buffer, then the untimed buffer conforms to the timed buffer.

6.2 Semantics for Timing

In this section, we describe how timing changes the execution and behavioral semantics of orbital nets. We will not yet define the type of value used for time, but we will assume that it is some numeric type that supports addition, subtraction, zero, and comparison operations.

We extend our untimed orbital net formalism with a function *treq* that maps places to timing requirements. This function returns the timing annotations, such as those in figure 47. Each timing requirement is a triple $(min, max, type)$, consisting of a minimum and maximum time value and a timing type. The minimum value is a nonnegative rational value. The maximum value is a rational value greater than or equal to the minimum value, or the special value ∞. The timing type is one of *behavior* (usually preceding transitions labeled with output or internal actions), *constraint* (preceding transitions labeled with input actions), or *untimed* (indicating that the minimum and maximum values have no meaning for this place). The range given by a timing requirement is the set of all time values t such that $min \leq t \leq max$.

In addition, a subset of the edges in the orbital net is specified by the set *ignore-timing*. This set is a subset of $P \times T$, and it indicates that specific transitions ignore the lower bound on a specific timing constraint. This extension is useful for consuming a token before its maturation age; for instance, it is used when implementing a time-out or watchdog timer.

For the purpose of generating traces, we assume a global clock (which we will call *time*) that takes on time values. In the initial state of the orbital net, this clock has a value of zero. Transition firings are assumed to be instantaneous; any number of transitions can fire without time advancing. This clock has no purpose other than generating timed traces and is not part of the state of the orbital net.

The marking of a net is extended as follows: Each token in the marking that resides in an *untimed* place remains as before. Each token in a marking that resides in a *constraint* or *behavior* place has a time-valued *age* parameter associated with it. Informally, the *age* parameter advances with time and describes how long the token has resided in the place.

We define the function *max-advance* on a marking that returns a time value. This function represents the maximum amount of time that can advance without allowing the age of some token in a behavioral place to exceed the maximum age allowed by the range. This function is defined as the minimum value of $(max - age)$ for all marked behavioral places, or ∞ if there are no marked behavioral places.

The original enabling condition for transitions in the orbital net is referred to hereafter as the untimed enabling condition. The timed enabling condition of transitions in the orbital net is defined by extending the untimed enabling condition as follows: A transition is enabled to fire if it is untimed-enabled to fire, and, in addition, all input places that have a behavioral timing requirement and that are connected to the transition by an edge that is not in the *ignore-timing* set must have a token with $min - age$ less than or equal to the *max-advance* of the marking. That is, we only allow a transition to be enabled if its behavioral places contain tokens that can become of age without causing some other token to exceed its maximum age.

The firing semantics of transitions is extended in the following way: Before the transition fires, time is advanced some arbitrary nonnegative amount greater than or equal to $min - age$ of each behavioral place input to the transition and less than or equal to *max-advance*. Time is advanced by increasing the value of all of the ages of tokens in timed places by precisely the same amount. If we are generating a firing sequence and thus need to compute the *time* value, we also increase it by exactly the same amount. Next, the constraint places in the entire net are checked. If any constraint place contains a token whose age is greater than the maximum value allowed by that constraint place, this firing is marked as a failure. Next, the tokens in the input places are removed as before, but tokens from *constraint* places that are connected with arcs not in *ignore-timing* are checked for timing violations. Specifically, if the age of the token being removed does not lie within the range specified by the timing requirement, then the firing is marked as a failing firing. (Since the previous step already checked the upper limits, this step only need check the lower limits.) Tokens are put into the output places as before, but all tokens put into places that are not of type *untimed* are assigned an *age* of zero.

We define a condition that, if not satisfied after the firing of a transition, indicates that the firing is a failing firing. This condition is that every behavioral place have an output transition that is untimed-enabled in the new state. This requirement ensures that every token in a behavioral place be consumable in all states in which its timing conditions are met, and thus the age at which the token is consumed cannot be controlled by external state.

Firing sequences are extended in the obvious way; rather than using a sequence of steps, we use a sequence of pairs of steps and timing values, where the timing value is the value *time* after time was advanced and before the tokens were moved. We define a function *untime* that takes a timed firing sequence and returns the untimed firing sequence created by removing the timing information.

We intend that the untimed net structure and behavioral place parameters determine the firing sequences allowed by the net, and that the untimed net structure and constraint place parameters determine the success or failure property of a firing sequence.

6.3 Receptiveness, Mirroring, and Multiple Timed Places

The distinction between constraint timing requirements and behavioral timing requirements preserves the correspondence between the constructions and behavioral properties of composition, receptiveness, and mirroring. Constraint and behavioral timing ranges are distinguished only by whether any transition immediately following the place (before composition) is labeled with an input action. If any transition is so labeled, then the place is marked as a constraint place; otherwise, it is marked as a behavioral place. Thus, a transition labeled with both an input and an output is treated as though it were only labeled with an input for the purposes of differentiating constraint and behavioral places.

The construction we use for composition works without change in the presence of timing. The net construction for receptiveness also remains exactly the same, because receptiveness is only used on input events, and timing constraints before input events are restricted to constraint places, which do not affect enabledness.

We retain mirroring in exactly the same way. If a specification is given with no unlabeled or hidden transitions, then the mirror exchanges the input and output action sets. It also switches constraint places and behavioral places—each constraint place becomes a behavioral place and each behavioral place becomes a constraint place in the mirror. This is intuitively reasonable when we consider constraint places as acceptors and behavioral places as generators. A problem arises, however, if a specification has multiple constraint places on a particular event.

A fair amount of attention has been given in the literature to forms of timed Petri nets where multiple timed places are inputs to a single transition (or synchronized transition). With our semantics, multiple constraint places are no problem (all constraint places must be satisfied) but, because of the way behavioral places are required to have an output transition that is untimed-enabled in all states where that place is marked, any place with multiple behavioral input places might cause unintended failures.

For the purposes of this dissertation, we find it sufficient to only allow a single behavioral place to be input to a transition or synchronized set of transitions. We are able to model everything we need with this restriction; without it, significant complexities arise. In particular, the liberal and conservative approximations fail to hold if the firing time of a behavioral place can be influenced by outside actions or by other behavioral places; we find it necessary to make the firing time independent of outside influences.

This restriction is not as severe as it might seem. Behavioral places only occur on outputs, and only a single circuit component is allowed to generate each output. All other elements see those events as inputs, so the places that are inputs to those transitions are interpreted as constraint places. Thus, restricting nets to a single behavioral place and multiple constraint places for a single transition reflects the natural characteristic of a single sender and multiple receivers of events on a wire in a circuit.

On the other hand, this is a restriction on mirroring, since we still allow multiple constraint places. We do not allow the mirror of such a description to be taken; at some point this shortcoming will need to be addressed. In the practical examples we have considered, taking the mirror of such a specification has not been necessary.

If a particular net will only be used as a specification, and thus we will be taking the mirror of it, then we allow multiple behavioral places preceding an action, since the mirror will turn the behavioral places into constraint places. We can do this because all cases where multiple behavioral places conflict are failures, and failure exclusion would eliminate these from our specification. This becomes useful when we check or measure several properties of a timed circuit, as we shall see.

We make no attempt to normalize a timed specification; any such specification must be provided in normalized form. This restriction has not yet posed any difficulties in practice.

6.4 Desired Properties of Timing Types

In order to implement these ideas in a computer program, we must consider the type of values that can be used for timing parameters and event occurrences—whether we restrict ourselves to small integers, larger integers, the rationals, or to the full set of reals, for each of these. But before we get too involved in theory, let us step back and discuss the types of systems we are modeling and the requirements of circuit analysis.

First, let us consider timing parameters. Timing parameters usually come from data sheets, circuit analysis (perhaps using Spice models), or protocol requirements; they are usually specified as decimal numbers with between one and four digits of precision. Thus, we dismiss any requirement for irrational timing parameters—irrational numbers only make sense when dealing with an infinite amount of precision. We will later see that using only rational numbers for timing parameters allows a finite representation of the infinite reachable state space.

The difference between a finite number of known rational numbers and the integers is a matter of scaling; instead of working in nanoseconds, we work in picoseconds or femtoseconds to avoid fractions. In general, we multiply all timing parameters in a system by the least common multiple of the denominators of the rational fractions to promote all constants to integers. We therefore eliminate any need for rationals through this type of promotion, and just consider integers.

On the other hand, we often have a couple of digits of precision, so we must not be restricted to small integers, unless we cannot handle larger ones. In this chapter, we will discuss techniques that require the use of small integers, and we will discuss some approximation techniques that relate systems parameterized by large integers are approximated by systems with small integers. In the next chapter, we will present other techniques that allow us to use more precise numbers without approximating the behavior of the system.

For event occurrences, we want to make as few assumptions as possible. When we describe a gate as having a delay of between one and two nanoseconds, we do not mean it has a delay of either exactly one or exactly two nanoseconds, but rather any value from the entire dense range. Prior research has discussed the difference between systems where

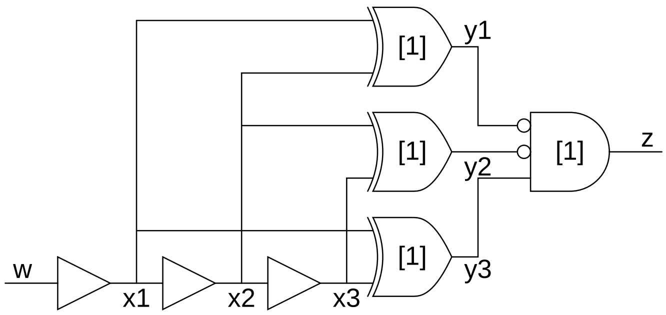


Figure 48. A circuit that illustrates the difference between synchronous, discrete, and continuous time.

events are constrained to occur on specific clock ticks, or at specific time values such as the integers, and systems where events can occur at any point within some dense domain [19]. For example, consider the circuit in figure 48, presented by Brzozowski and Seger [10] and slightly modified here. This example illustrates that for some models and interpretations, it is essential to consider time more accurately than the precision of the timing parameters themselves. Initially, all wires are low. At time zero, a rising edge occurs on w, and w remains high for the rest of time. The three buffers have arbitrary delay, so we see a sequence of rising edges on $x1$, $x2$, and $x3$, in that order, but at otherwise arbitrary times.

The four remaining gates have a delay of exactly one, and they ignore pulses shorter than one. The top exclusive-or gate measures the difference between the times of the edges on $x1$ and $x2$; if the difference is greater than one time unit, then $y1$ will go high for an amount of time equal to the difference between the occurrences of edges on $x2$ and $x1$. Similarly, the second exclusive-or-gate measures the difference between the edges on $x2$ and $x3$, and the third exclusive-or gate measures the difference between $x3$ and $x1$. The and-gate will fire if and only if there is a period of time greater than one unit of time during which the first two exclusive-or gates are low, and the third is high. We want to know if z can ever go high.

We will show that z can indeed rise using the continuous time model. The following timed event sequence has the three rising edges on $x1$, $x2$, and $x3$ close enough together so that neither of the first two exclusive-or gates fire, but the first and last edge are far enough apart such that the third does indeed fire.

$$[(x1+, 0.2), (x2+, 0.8), (x3+, 1.4), (y3+, 1.2), (z+, 2.2), (y3-, 2.4), (z-, 3.4)]$$

Now, we consider discrete time. We show that, using a time step of one unit of time and the defined behavior of the primitive gates, z can never rise.

Let us slightly refine our definition of the behavior of a gate with delay; if the gate is enabled to change for exactly one time unit, then it will change and will remain high for exactly one time unit. This specification of a behavior depending on an infinitely accurate measurement of some interval of time means little in the physical world, since no measurement of time is absolutely accurate. On the other hand, the synchronous time model allows us to observe events only on these precise time boundaries, so we must define what happens there.

With this type of model, if z goes high at time t, then $y3$ must be high and $y1$ and $y2$ low during the previous step. This implies that, during the preceding step, $x1$ and $x2$ have the same value, $x2$ and $x3$ have the same value, but $x1$ and $x3$ have different values. This is obviously not possible, so in the synchronous time model, z can never go high.

If we say that a gate enabled to change for exactly one time unit does not change, the same type of analysis holds.

If we use a time step of less than one unit of time (for instance, 0.2 units), then we can indeed find a trace for which z rises; the continuous trace above is one such example. In general, for many properties and circuits, there is some k such that a time step of $1/k$ will give the correct answer. No general algorithm for calculating this k is known, and Alur presents in [1] a system and property for which there is no k that will yield the correct answer.

If we allow the circuit to nondeterministically choose whether a pulse of width exactly one is ignored or is turned into a transition, then we find a discrete trace for which the circuit raises the z wire. We only allow such a choice to be made if we know that the input pulse will only last a single time unit, because it is enabled to change in the current step. We justify such a behavior by observing that, if we cannot determine whether the separation between two events is less than or greater than a particular value, then we must consider both case possible . The following trace suffices:

$$[(x1+, 0), (x2+, 0), (x3+, 1), (y3+, 1), (z+, 2), (y3-, 2), (z-, 3)]$$

For this particular trace, the bottom two exclusive-or gates are enabled for precisely one time unit. The middle one chooses not to fire; the bottom one chooses to fire, and z rises.

Orbital nets can only describe behaviors like this last one; because of this, we will be able to prove that, in our formalism, restricting events to occur at discrete times yields exactly the same results as a continuous time analysis. Indeed, we will prove that this is true for a large class of transition systems, including most forms of timed Petri nets, where the properties of interest meet certain requirements.

This is a strong result. In general, continuous time verification is much harder than discrete time verification. By proving that all models and all properties described by our orbital nets yield the same answers under discrete time and continuous time semantics, we allow the use of simpler and faster analysis.

We do not permit the constraints or enabledness of transitions to be based on whether clock values are less than or greater than a particular constant, but only allow less than or equal to and greater than or equal to. Thus, a discrete range includes both end points and covers the range. If less than or greater than is allowed or can be generated by negating a $\leq$ or $\geq$, then the discrete model does not cover the range; there is an area near an endpoint of the range that cannot be considered under the discrete model.

We now formalize this by proving that, for orbital nets, we only need to consider integer event occurrence times in order to get an accurate answer for continuous time behavior.

6.5 Discrete Approximations of Continuous Time

In this section, we show that considering only discrete values gives us an exact analysis, even for the continuous time case. The proof we will supply is more general than required for our specific semantics; we also show that this result holds for a large class of timed transition systems, including most versions of timed Petri nets and timed automata systems limited to disallow strict less than or strict greater than. We shall start by stating the conditions that the timed transition system must satisfy in order for our result to apply.

First, the system must be a transition system extended with a set of clocks, where the clocks can take values from the nonnegative reals. The clock values are considered the timed state; the untimed state is the full set of other values in the system. Each transition can be enabled by a guard condition. The guard condition is allowed to be the conjunction of an untimed guard and a timed guard. The untimed guard can be any general predicate on the untimed state of the system. The timed guard must take on one of the following forms:

$$\begin{aligned} \textit{timed-guard} = \textit{true} \mid \textit{false} & \\ & \mid \textit{clock} \leq \textit{integer-constant} \mid \textit{clock} \geq \textit{integer-constant} \\ & \mid \textit{timed-guard} \vee \textit{timed-guard} \mid \textit{timed-guard} \wedge \textit{timed-guard} \end{aligned}$$

The integer constants in these expressions must be nonnegative. When transitions fire, they may reset any subset of the clocks to zero. In the initial state of the transition system, all clocks are assumed to be zero. In addition to firing transitions, such a transition system also has an operation to advance time. This operation may take place between any transition firings, and it may occur any number of times in succession. Time is advanced some nonnegative amount t by adding t to all of the clock variables. There may be a limitation on how much time can advance in any particular state. This limitation is expressed in the same form as a timed guard; time may not advance such that the timed guard is false at the point to which time is advanced. We require that this guard be composed of strictly a conjunction of upper bounds on the clocks so that it becomes false monotonically.

For any such transition system, we define a global clock *time* in the same way we have for our orbital nets. We define a firing sequence as a sequence of pairs of transitions and times, the latter taken from the value of *time* when the transition fired. We can derive the value of all clocks at any point during the firing sequence from the firing sequence itself, since we know precisely which transitions reset which clocks and the occurrence time of each transition.

In addition to these operational aspects of the net, we also define properties of firing sequences. For instance, the primary property of orbital nets is failure-freedom. Any property of a firing sequence has two components consisting of an untimed component and a timed component. The untimed component may be any computable function of the untimed firing sequence. The timed component is expressed in terms of the conjunction of specific property functions for each transition in the firing sequence. Each such specific property function is defined in terms of the values of the clocks immediately before the transition is fired and can take the following forms:

$$\begin{aligned} \textit{timed-property} = {} & \textit{true} \mid \textit{false} \\ & \mid \textit{clock} \leq \textit{integer-constant} \mid \textit{clock} \geq \textit{integer-constant} \\ & \mid \textit{timed-property} \wedge \textit{timed-property} \end{aligned}$$

We do not allow timed properties to use disjunction.

This description of a timed transition system is very general. It can describe the behavior of orbital nets, time Petri nets, timed Petri nets, many variations of Petri nets with timing ascribed to the places, and timed automata (if the conditions on the guards are satisfied). It can also model many properties, including the failure function used in orbital nets, general untimed properties such as deadlock or reachable state properties, and virtually all timing properties that do not use disjunction or negation.

Let us consider more carefully how the orbital nets formalism fits this description. Since orbital nets are a transition system, the untimed properties are obviously satisfied. The semantics that has been used for orbital nets combines time advancement and transition firing into a single operation, but these are easily separated into two different operations and the enabling condition redefined appropriately. The token ages are the clock variables. The limitation on how far time can advance in a particular timed state is the conjunction of the constraints, over all marked behavioral places, that the token ages cannot exceed the maximum allowed age. The only remaining area of difficulty is defining the failure set in terms of our restrictions on properties. The untimed aspects are trivial: deadlocks, markings containing places with no output edges, and receptiveness failure transitions are all untimed properties of the untimed firing sequence, as is the existence of a marked behavioral place that does not have an output transition that is untimed-enabled. Because of this restriction on the behavioral places, no deadlock can be caused by timing requirements in places. There are two remaining failure conditions: consuming a constraint place too early and allowing a constraint place to age too much. These are handled by associating a timed property predicate with each transition; this predicate is the conjunction of $marked \rightarrow age \leq max$ for each constraint place in the net, and $min \leq age$ for each constraint place token consumed by that particular transition firing. Thus, success can be described as a *timed-property* within the above restrictions.

We define a discrete firing sequence to be one in which all events occur at integer time values. A continuous firing sequence is one in which the event occurrences are not restricted in this manner. Obviously, the former is a subset of the latter. We will define a function $f(\alpha, r)$ that takes a continuous firing sequence α and a real value $0 \leq r < 1$ and returns a discrete firing sequence, such that both α and $f(\alpha, r)$ have the same underlying untimed firing sequence.

The function $f(\alpha, r)$ is defined recursively as follows:

$$f(\epsilon, r) = \epsilon$$

$$f((tr, t)\alpha, r) = (tr, \lfloor t + r \rfloor) f(\alpha, r)$$

We add r to the time of each event occurrence and then take the floor of the resulting value. Obviously, the untimed firing sequence remains the same. Each discrete firing sequence maps to itself.

We start with a few simple lemmas.

Lemma 6.1. For every value a and b and integer n, if $a - b \leq n$ then $\lfloor a \rfloor - \lfloor b \rfloor \leq n$. In addition, if $a - b < n$, then there exists an r such that $0 \leq r < 1$ such that $\lfloor a + r \rfloor - \lfloor b + r \rfloor < n$.

Proof. The first part of the proof is straightforward.

$$a - b \leq n$$

$$a - n \leq b$$

$$\lfloor a - n \rfloor \leq \lfloor b \rfloor$$

$$\lfloor a \rfloor - n \leq \lfloor b \rfloor$$

$$\lfloor a \rfloor - \lfloor b \rfloor \leq n$$

We choose $r = \lceil b \rceil - b$. Obviously, $0 \leq r < 1$. Also, $b + r = \lceil b \rceil$, which is an integer. Finally,

$$\begin{aligned} a - b &< n \\ a + r &< n + b + r \\ \lfloor a + r \rfloor &< \lfloor n + b + r \rfloor \\ \lfloor a + r \rfloor - \lfloor b + r \rfloor &< n. \end{aligned}$$

These two general properties of inequalities and the floor function will prove useful in later lemmas. ∎

Lemma 6.2. Every discrete firing sequence is also a continuous firing sequence with the same timed properties.

Proof. Since each integer is also a real value, this lemma is immediate. ∎

Lemma 6.3. For every continuous firing sequence α, and every value of r such that $0 \leq r < 1$, $f(\alpha, r)$ is a legal discrete firing sequence also generated by the transition system.

Proof. The firing sequence $f(\alpha, r)$ is obviously discrete. It is legal if it is a member of the set of discrete firing sequences.

Let us assume that it is not a legal firing sequence. We need only consider the smallest prefix of the firing sequence that is not a legal firing sequence; if there is no such prefix, then the firing sequence must be legal. The last transition in this firing sequence must not have been enabled to fire at the time it fired, or else it was not legal for time to advance to the time at which this transition fired. We start with the first possibility.

All of the untimed enabling conditions for this transition are met, since this transition was enabled in the original continuous timing sequence. Thus, the timed guard for this transition must be false in the discrete firing sequence while it was true in the original continuous firing sequence. We show that every timed guard true in the original sequence must also be true in this new discrete sequence.

If the timed guard is *true* or *false*, then this holds trivially. If the timed guard is a conjunction or disjunction of two timed guards, this holds by induction on the component timed guards. Thus, there must be at least one timed guard of the form *clock* $\leq$ *integer-constant* or *integer-constant* $\leq$ *clock* that is true in the continuous firing sequence but is not true in the discrete firing sequence.

We will define the time that the last transition in the firing sequence occurred as t_2. We define t_1 as the time of occurrence of the last transition in the immediate prefix of the firing sequence that reset *clock*; if there is no such transition, we set t_1 to zero to reflect the initialization of the clocks. Thus, in the original firing sequence, the value of *clock* at time t_2 is $t_2 - t_1$. In the discrete firing sequence, the value of *clock* at the time that the last transition was fired was $\lfloor t_2 + r \rfloor - \lfloor t_1 + r \rfloor$. But if, in the original sequence, $t_2 - t_1 \leq k_1$ for some integer constant k_1, then by our first lemma $\lfloor t_2 + r \rfloor - \lfloor t_1 + r \rfloor \leq k_1$, so the timing guard is also true in the discrete sequence. In addition, if $k_2 \leq t_2 - t_1$ for some integer constant k_2, then by our first lemma $k_2 \leq \lfloor t_2 + r \rfloor - \lfloor t_1 + r \rfloor$. So any timing guard that is true in the original continuous firing sequence must also be true in the discrete firing sequence.

The only other way that the firing sequence might not be legal is if time cannot advance to t_2 in the state reached by the immediate prefix of the firing sequence. Yet, because the time advance function is just another timing guard, if it was true at t_2 in the original firing sequence, it must also be true at $\lfloor t_2 + r \rfloor$ in the discrete firing sequence.

Therefore, by contradiction, the lemma must hold. ∎

Lemma 6.4. For every continuous firing sequence α, and every value of r, all timing properties that are true of α are also true of $f(\alpha, r)$.

Proof. Every timed property is a conjunction of expressions in the form of timed guards. Since every possible timed guard true in the original firing sequence is also true in the discrete firing sequence, every true timed property for each transition in the original firing sequence is also true in the resulting firing sequence. Thus, every true timed property of the entire original firing sequence is also a true timed property of the discrete firing sequence. ∎

Lemma 6.5. For every continuous firing sequence α and every timing property p that is false of α, there exists some r such that the timing property p is also false in $f(\alpha, r)$.

Proof. Let us examine the timed property that must be false. We first select any particular transition firing that caused this property to be false. If the timed property expression for this transition is a conjunction, we select one of the subexpressions that is false in the continuous firing sequence; there must be one such subexpression. If it is the constant *false*, then the timed property is trivially false in the discrete case. Otherwise, it must be an expression of the form *clock* $\leq$ *integer-constant* or *integer-constant* $\leq$ *clock*. As before, we define the time at which this transition fires as t_2, and the time of the most recent resetting of the clock as t_1. Since the property was false in the original firing sequence, we have either $t_2 - t_1 >$ *integer-constant* or $t_2 - t_1 <$ *integer-constant*. Yet, according to the first lemma, we can choose an r such that $\lfloor t_2 + r \rfloor - \lfloor t_1 + r \rfloor >$ *integer-constant* in the first case or $\lfloor t_2 + r \rfloor - \lfloor t_1 + r \rfloor <$ *integer-constant* for the second. Thus, we can choose an r such that a single property that is false in the continuous firing sequence α is also false in the discrete firing sequence $f(\alpha, r)$. ∎

We now explain why timed properties have been restricted to allow only conjunction and not disjunction. Each timed property requires us to choose a specific r to generate a discrete timing sequence with that same failing property. If we were allowed to use disjunction, we might need to find a single r that makes both subexpressions false, and this might not be possible.

Theorem 6.1. For a given timed transition system and timed property, there exists a continuous firing sequence with that property if and only if there exists a discrete firing sequence with that property, and there exists a continuous firing sequence without that property if and only if there exists a discrete firing sequence without that property.

Proof. The forward implication of the first clause is lemma 6.4, and the forward implication of the second clause is lemma 6.5. For both clauses, the reverse implication is shown by lemma 6.2. ∎

For our purposes, if there is no discrete firing sequence that is a failure, there is also no continuous firing sequence that is a failure. Thus, we consider only discrete event occurrence times in our verification of a timed orbital net. Therefore, we almost have a finite state space when generating the reachable states of an orbital net. We have restricted our clock values to the integers, but there is no restriction on the maximum ages of the clocks. The next section shows how we normalize each timed state and discusses the size of the resulting state space.

6.6 Normalization

The set of values that clocks are compared against is well defined, and once a clock exceeds the largest value it is compared against, the amount by which it exceeds that value is no longer of interest. Alur makes the same observation in [1]. We can *normalize* each state by replacing each clock value that has exceeded the maximum value it is compared against with that value plus one. Once we have done this, we have a finite representation of our reachable states.

The size of the state space is still large, however. We estimate how large it is by considering how many clocks can be active at a given time, and what values they can have. In our orbital nets, there cannot be more clocks than there are orbits. Let us call the size of our untimed state space $|S|$, the number of orbits n, and the maximum integer value used in any constraint or behavior place k. Then, an upper limit on the size of our timed state space is

$$|S|k^n.$$

It is instructive to compare this with other techniques for representing continuous time. Alur, in his dissertation [1], shows that any timed transition system with clocks (and allowing negation and $<$) has an upper limit on its state space, when using the unit cube method, of

$$|S|(2k+2)^n n!.$$

A more accurate estimate is

$$\frac{1}{\ln 2} n! \left(\frac{k}{\ln 2}\right)^n 4^{1/k}.$$

(For $n = 10$ and $k = 10$, this gives a result of $2.3490321485322 \times 10^{18}$, which is correct to ten significant digits.) Thus, by using a discrete approximation, we save more than $n!$, which is super exponential. (All other multipliers are greater than one.) The main requirement is that we do not use $<$ or $>$, but only the unstrict $\leq$ and $\geq$. Also, we do not allow negation. As we have shown, neither of these limitations affect our ability to describe circuits.

6.7 Conservative and Liberal Approximations

Nonetheless, even this smaller state space can be large. If we promote three digit numbers to integers, k is usually on the order of 300 or so. Circuits have n proportional to the number of active components. In this section, we discuss the *conservative* and *liberal* approximations to an orbital net that can be verified using a much smaller state space. The technique we use is similar to the way we scale up rational numbers to integers. Let us imagine that we wish to scale the rationals by a value less than that required to make all of the parameters rational, and then convert the resulting rational numbers to integers such that we still get meaningful results from verification.

If all of the timing parameters are originally integers, and if they have a greatest common factor that is greater than one, then we may divide all of the parameters by this greatest common factor without affecting the accuracy of our verification at all.

This is made possible by the way behavioral place parameters determine the behavior of the net, and constraint place parameters determine the success or failure of a particular firing sequence.

For any positive real value d, we define two functions on timing requirements as follows:

$$conservative(min, max, type) \equiv (\lceil min/d \rceil, \lfloor max/d \rfloor, type)$$
$$liberal(min, max, type) \equiv (\lfloor min/d \rfloor, \lceil max/d \rceil, type)$$

Note that *conservative* may return a malformed timing requirement if $\lceil min/d \rceil > \lfloor max/d \rfloor$.

A conservative approximation to our orbital net is given by applying the *conservative* function to each timing requirement of type *behavior*, and applying the *liberal* function to each timing requirement of type *constraint*. (If any place now has a malformed timing requirement, we cannot construct a conservative approximation for the net using the current value of d.) Informally, this net has a reduced set of firing sequences (even though now all timing requirements have integer values, we are still treating them as real values for the purposes of analysis). Thus, if this net has any failing firing sequences, so does the original net.

A liberal approximation to our timed orbital net is given by applying the *liberal* function to each timing requirement of type *behavior*, and applying the *conservative* function to each timing requirement of type *constraint*. We must check that we have not generated any invalid timing requirements, as before. Informally, this net has a larger set of firing sequences. Thus, if this net has no failing firing sequences, then neither does the original net.

So, we choose values of d that give us small integers for the values of the timing requirements and explore the search space for both the liberal and the conservative approximations to the net. If the liberal approximation succeeds, or the conservative approximation fails, we are finished; otherwise, we must choose another value of d that gives us slightly larger integers, and proceed.

Therefore, if $Q(N)$ is a function denoting the set of all firing sequences of net N (referred to as the language of the net), we have $Q(\mathit{conservative}(N)) \subset Q(N) \subset Q(\mathit{liberal}(N))$.

If two nets N_1 and N_2 are identical except for the timing requirement in a behavior place p, and $\mathit{treq}(p_1)$ has a range that is a superset of the range of $\mathit{treq}(p_2)$, then every firing sequence in the language of N_2 is also in the language of N_1. Extending this inductively, if every behavioral place of N_1 has a *treq* with a range that is a superset of the corresponding behavioral place of N_2, then the language of N_1 is a subset of the language of N_2.

Because the constraint places are unchanged, firing sequences in both languages can only be turned into failures from successes or into successes from failures because of deadlocks introduced when we prune away the successors to a particular firing sequence. On the other hand, untimed enabledness of behavioral places is required for all marked behavioral places in any net in any state. For this reason, if there is a behavioral place ready to fire in the original net, it must also be ready to fire (although perhaps at a slightly later time) in the second net. Thus, when we reduce the language by reducing the range of the behavioral timing requirements, we do not introduce any new deadlocks.

In summary, applying the *conservative* function to the timing requirements of the behavior places of a net reduces its language, and applying the *liberal* function to the timing requirements of the behavior places of a net increases its language.

Now consider what firing sequences in a language are failures. Deadlock failures are strictly a function of the language (they are those sequences that are not a prefix of some other sequence in the language). All other failures that are not untimed failures are due to the violation of the requirements of a constraint place.

If two nets N_1 and N_2 are identical except for the timing requirement in constraint place p, and $treq(p_1)$ has a range that is a superset of the range of $treq(p_2)$, then every successful firing sequence in N_2 is also a successful firing sequence in N_1. Since their languages are identical, every failing firing sequence in N_1 is also a failing firing sequence in N_2. An inductive argument extends this for any number of constraint places in a net.

Therefore, consider the construction of a conservative net as two steps. We first apply the *conservative* function to all of the behavior places, resulting in a new net with a smaller language. This will introduce no additional failures. Then, we apply the *liberal* function to all of the constraint places. (We only do this to get into the realm of integers, where we use our integer approximation.) This will introduce no failures. Thus, any failure that exists in the conservative net also exists in the original net.

Now consider the liberal net. Applying the *liberal* function to all of the behavior places results in a new net with a larger language; no existing failures are turned into successes. Applying the *conservative* function to the constraint places will hide no failures. So, if no failures exist in the liberal net, then no failures exist in the original net.

6.8 Specifying Timing Information

Since the input form of our representation is textual, we need a method for attaching timing information to places. Since we have a current place at every point during the evaluation of an expression, it is trivial to add a command to annotate the current place. The `delay` primitive provides this capability.

`(delay` *time-spec*`)` Attaches a timing specification to a particular place. The *time-spec* must start and end with a colon, and it can take one of the following forms:

`:#:` If only a single value is provided, the minimum and maximum values are the same.

`:#,#:` If two values are given separated by a comma, then they are used as a minimum and maximum value, respectively.

`::` If no values are given, this indicates a place that is removed after the net is built. This variant was described in chapter three.

The `delay` keyword and parentheses are optional, since any lone symbol that ends with a colon is automatically treated as if wrapped in a `delay` call.

The values can be either nonnegative numbers in decimal format, the special indicator `inf` or `infinity`, general variables, or general variables followed immediately by an asterisk and then a decimal number. Variables used in timing specifications are called timing variables. When followed by an asterisk and a decimal number, the value of the variable is multiplied by that decimal number; thus, it is possible to specify different delays that scale in a uniform manner. Timing variables must be assigned (with `def`) to either a number in decimal format, the value `infinity`, the value `ignore`, or the value `unknown`. The value `ignore` evaluates to either 0 or `infinity` depending on whether the timing variable is used to specify the minimum or maximum of the range. The value `unknown` indicates a timing unknown that the system solves for. This value can only be used on behavioral timing constraints; a maximum of one such variable is allowed, and the variable must set either the minimum or maximum (but not both) of the range. If the description has such a variable, then the `verify` procedure will use a simple binary search technique to determine the value of the variable that maximizes the range without yielding a failure.

Places that will be merged away (usually by invoking labels from `let` commands as transfer of control) cannot be annotated with time information. In particular, this means that delays must occur at the beginning, rather than at the end, of loops.

(We originally used brackets rather than colons, but the Scheme compiler we use does not allow brackets in symbols it reads. Thus, while colons are used in the input language, square brackets are used in diagrams and everywhere else.)

The `delay` primitive does not indicate whether a place is a behavioral or a constraint place. That information is derived from the structure of the net and the assignment of input or output wires for the models.

6.9 Timing Examples

By now we have discussed enough theory. Let us turn to some examples. The first example we consider is the simple timed buffer. We use the model from the first figure in this chapter, but remove the timing constraints on the input events.

```
(def (timed-buffer (a) (b))
   (process a+ :2,10: b+ a- :2,10: b-))
```

The first question is, does this buffer conform to an untimed buffer?

```
(verify
   (with-wires global ((a 0) (b 0))
      (timed-buffer a b)
      (rec-buffer-mirror b a)))
```

The answer is, yes, it does; this is as we would expect. But does an untimed buffer conform to this timed buffer?

```
(def timed-buffer-mirror (mirror timed-buffer))
(def timed-buffer-spec (receptive timed-buffer-mirror))
(verify
   (with-wires global ((a 0) (b 0))
      (buffer a b)
      (timed-buffer-spec b a)))
```

The result indicates that it does not, due to a constraint violation; the untimed buffer can generate outputs both before and after the timed buffer allows. Do two timed buffers with delays of `:1,5:` conform to a single timed buffer with delays of `:2,10:`?

```
(def (timed-buffer2 (a) (b))
   (process a+ :1,5: b+ a- :1,5: b-))
(verify
   (with-wires global ((a 0) (b 0) (c 0))
      (timed-buffer2 a b)
      (timed-buffer2 b c)
      (rec-buffer-mirror c a)))
```

This succeeds. We cannot check if a single buffer conforms to a pair of faster buffers because the pair has an internal wire on which hidden events occur. If we were not using timed nets, we could normalize the specification, but we cannot normalize a timed specification.

A single timed buffer does not conform to two timed buffers with half the delay because the internal wire acts as a storage node; the composed buffers successfully accept a larger set of inputs than the single slower buffer. Thus, it is not always safe to replace a series of buffers with a single buffer having a longer delay.

The buffer definition we have used so far does not allow any input changes when an output change is pending. Thus, for circuits in which hazards must not occur on any wire, it is an appropriate model to use. However, many timed circuits have hazards and operate correctly in the presence of these hazards. Therefore, we would like to be able to define a timed model of a buffer that allows an input hazard, and responds appropriately. In the previous chapter we defined an untimed buffer that ignores hazards; it is easy to adapt this for the timed case. For the timed case, however, we must use the special `ignore-timing` feature of our nets, since an input event that disables an output event must clear the timed behavioral place, perhaps before the behavioral place is ready to fire. The primitive `ignore-timing` allows us to specify this in our model.

`(ignore-timing)` This primitive function declares that the timing before the previous transition is ignored for the purpose of the previous transition. Thus, the

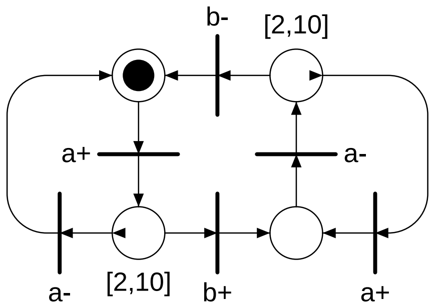


Figure 49. A timed hazard-ignoring delay buffer. The arrow inside the place indicates that the respective edge ignores the timing.

`(ignore-timing)` invocation goes *after* the event that it pertains to. This awkward placement is necessary because the edge that is annotated does not exist until after the transition is created; a revision of the language may work around this.

We also use the `case` statement, which is similar to `guard`, but lifts the restriction that the first element of each choice be a transition. The `case` statement can make state machines easier to read.

`(case` *expr branches*`)` This statement is much like the `switch` statement of C. The *expr* is evaluated; it must result in a constant integer at compile time. For this reason, `case` is most often used within `sm`. Each *branch* consists of a list; the first element must be an integer constant, and the rest is treated as the body of that branch. The `case` statement acts as a `choice` over all branches that start with the constant to which *expr* evaluates.

When we put these together, we end up with the following description. We only need timing on one of the branches in each case.

```
(def (timed-buffer3 (a) (b))
   (wire-init-set! (a 0) (b 0))
   (complete ()
      (sm loop ((s 0))
         (case s
            (0 a+ (loop 1))
            (1 :2,10: b+ (loop 2))
            (2 a- (loop 3))
            (3 :2,10: b- (loop 0))
            (1 a- (ignore-timing) (loop 0))
            (3 a+ (ignore-timing) (loop 2))))))
```

The resulting net appears in figure 49.

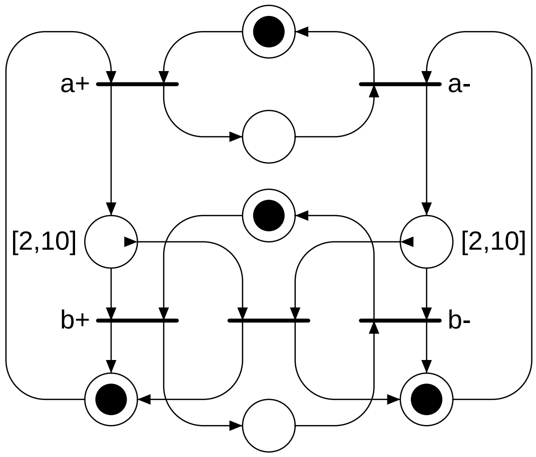


Figure 50. A model of a buffer that can nondeterministically ignore or pass forward a hazard. It can also store a pulse, much like a real buffer.

This net conforms to the previous delay model, which did not allow any hazards. The composition of two of these new models does not conform to a single occurrence of one with larger delay values, however; this is the first buffer model we have seen for which this is true. The failing trace (where a is the composite input, c is the composite output, and b is the internal wire) is $[a{+}, b{+}, a{-}, c{+}]$; this trace is not allowed a single buffer, but is allowed by the concatenation of two.

This net is similar to the untimed hazard ignoring net, except that every output event that occurs must be preceded by a causal input event satisfying the timing constraint. On the other hand, this buffer may not be realistic; in the real world, feeding a pulse into a buffer will often generate an output pulse where the leading edge of the output pulse comes after the trailing edge of the input pulse. This is because there are usually at least two different capacitive nodes and some slightly inductive wires in a buffer that can store a pulse and then later feed it out. This leads us to a third timed model for a buffer. This model allows a single hazard to occur and starts up independent clocks for the two separate edges. It also allows a hazard to disappear inside the buffer. We allocate a temporary wire that synchronizes the cancellation of two events.

```
(def (timed-buffer4 (a) (b))
   (with-wires ((t 0))
      (par
         (process a+ a-)
         (process b+ b-)
         (process a+ :2,10: (choice (t+ (ignore-timing)) (b+)))
         (process a- :2,10: (choice (t+ (ignore-timing)) (b-))))))
```

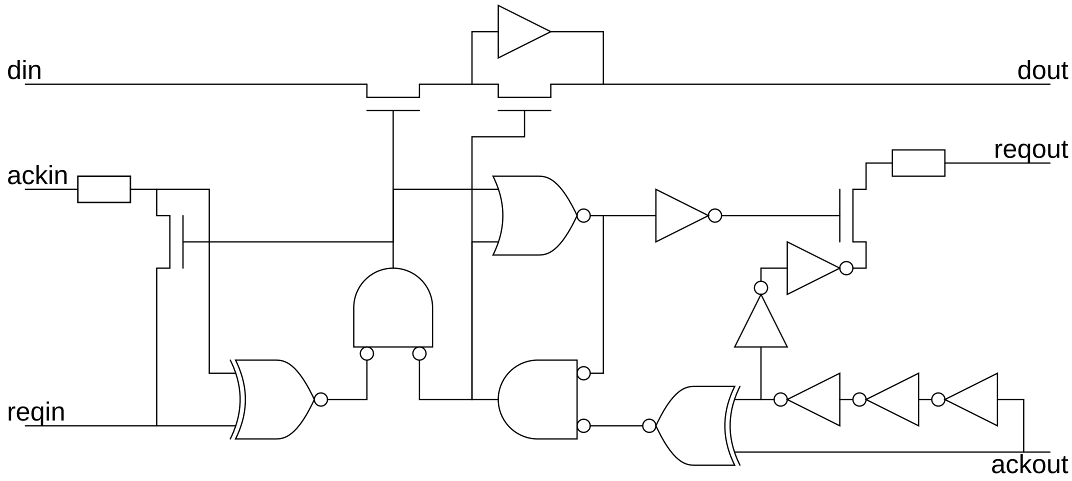


Figure 51. Seitz queue element

This buffer is shown in figure 50. Using it, we can verify circuits that have hazards, but which operate correctly in spite of them. This buffer does not assume that, if a hazard has a width of at least the maximum delay less the minimum delay of the buffer, then an output pulse will be generated; in general, this assumption does not hold for physical circuit elements. Instead, this buffer allows a nondeterministic choice between generating an output under these conditions, and ignoring the pulse; this is a more accurate and more conservative model than that used by Burch [11].

We use this model to analyze the Seitz queue element. The circuit is given in figure 51. It uses two-phase signaling on its input and output (meaning that there is no distinction between rising and falling edges, and each such edge indicates a request or acknowledge), so we use a two-phase specification and two-phase gates for the input and output signal paths. We will model the pass transistors as simple latches. Finally, since any hazard on the input and output signal paths will be propagated to the output, we know such hazards must be failures, and thus we combine the multiple delays on those paths into a single delay.

This circuit was originally designed by Seitz [32], who intended it to be correct under the 3/2 gate delay model—that is, the maximum delay of a gate is assumed to be less than 3/2 of its minimum delay. An error in the circuit was found by Mishra [37] and repaired by Browne, Clarke, Dill, and Mishra [9], and separately by Burch [11], whose circuit we will investigate. We will show how Orbits can automatically find the maximum timing range for which the circuit works correctly.

We model each gate as a perfect logic element followed by a delay. We specify the above circuit in our language using the methods that have gone before; we omit the details for brevity.

Now we present a specification for this queue element.

```
(def (queue-spec (reqin ackout din) (ackin reqout dout))
   (with-wires (inval outval)
      (par
         (process (choice (din+) (din-) (reqin+ ackin+)))
         (process (choice (dout+) (dout-) (reqout+ ackout+)))
         (process (burst ackin+ ackout+))
         (complete ()
            (sm loop ((s 0))
               (case s
                  (0 din+ (loop 1))
                  (0 reqin+ :: inval- (loop 0))
                  (1 din- (loop 0))
                  (1 reqin+ :: inval+ (loop 1)))))
         (complete ()
            (sm loop ((s 0))
               (case s
                  (0 dout+ (loop 1))
                  (0 reqout+ :: outval- (loop 0))
                  (1 dout- (loop 0))
                  (1 reqout+ :: outval+ (loop 1)))))
         (complete ()
            (sm loop ((s 1))
               (guard
                  ((and (> s 1) (odd s)) outval+ (loop (/ s 2)))
                  ((and (> s 1) (even s)) outval- (loop (/ s 2)))
                  ((< s 4) inval+ (loop (if (= s 1) 3 (+ s 4))))
                  ((< s 4) inval- (loop (if (= s 1) 2 (+ s 2))))))))))
```

First, we allocate two internal wires that are used to signal the value that has been read into or written out of the queue; a rising edge indicates a positive value, and a falling edge indicates a negative value. We have six concurrent processes. The first two allow the data input and data output to change only before a request or after the corresponding acknowledge; they also enforce two-phase signaling. The third process paces the input acknowledge to make sure that we do not store more than one data element inside the queue; if we have acknowledged two input data items before the receiver has acknowledged one, then we are responsible for storing two data items, and the circuit cannot do that. The next two processes signal an input or output value by keeping track of the value of the `din` and `dout` wire, respectively, and strobing the `inval` or `outval` wire when the appropriate request occurs.

The final process enforces the queue discipline; that is, the appropriate value must be written out by the queue. We represent the values stored in the queue by a single state constant; a value of one indicates that the queue is empty. As new values enter the queue, the most significant digit is moved over one place, and the new value takes its original position. Values are taken out of the queue from the least significant bit. We have to allow for two values to be stored, since we note the values at the request, and we might be storing one value while a request to store another comes in. Thus, our queue can take on state values of empty (one), storing a single value (two and three), and storing two values (four through seven).

We investigate which delay values allow the circuit to work. To do this, we specify the delay in the delay buffer as `:1,tmax:` where `tmax` is a timing variable that we define to be `unknown`. (For the buffers created from multiple gates, we use delays of `:2,tmax*2:` and `:3,tmax*3:`.) Then, we run the verification procedure as before.

The verification proceeds automatically, searching for the value of `tmax` that would create the largest timing range for which the circuit functions correctly. The program first tries using the minimum range by setting `tmax` to 1; this succeeds. Then it tries a maximum range by setting `tmax` to `infinity`; this fails almost immediately. Next, it tries a delay of 2 for `tmax`, and this also fails quickly. Now that it has a finite range of potential values, it narrows the range down: 1.8 fails, 1.6 fails, 1.4 fails, 1.3 fails, 1.2 fails (and the failure trace is longer with each successive failure). The next value, 1.1, works. It continues to refine the boundary in this fashion, trying 1.18, which works, then 1.19, which works, then 1.198, 1.199, 1.1998, and 1.1999, all of which work. It then reports that

```
1.1999 works and 1.2 fails.
```

The entire procedure takes under a minute.

Thus, even though we are using a more conservative model than that used by Burch, his results still hold; the circuit does not function correctly under the 5/6 model (this corresponds to a delay of `:1,1.2`), but it does for any lesser value.

Requiring the timing of all of the gates in a circuit to be within plus or minus 9% of each other (as 5/6 timing requires) is strict, even for such a small circuit. This is especially true when the requirement is for both rising and falling edges. For instance, a CMOS nor-gate has significant asymmetry in its delays for rising and falling edges and from the two inputs. The rise and fall time of the gate from one input might be 220 and 390 picoseconds, while from the other input they might be 560 and 490 picoseconds. The rise times will tend to differ more than the fall times for CMOS nor-gates, because the rise times rely on charging the output node through series transistors, and the two series transistors will have different gate-to-drain voltages and different amounts of source capacitance to charge; the parallel pulldown transistors will cause the fall times to be closer to each other. The converse is true for nand-gates. For complex gates, the variance in gate delays can be quite significant.

Thus, were the Seitz queue element implemented as shown, careful attention would have to be paid to layout, and a timing analysis based on the final layout would have to be performed. Another alternative is modifying the circuit, possibly by adding delay on some paths, to make it less sensitive to timing variances.

We now turn our attention back to buffers. The hazard model we presented is not receptive. It still does not accept three quick edges. The behavior of a buffer when it receives such an input is hard to define. We specify that under these conditions, the buffer enters *chaos mode* [11], in which it can generate any alternating output edges and accept any input. We also allow the buffer to leave chaos mode if its input is stable for a duration equal to the buffer delay. Otherwise, it performs just as our normal hazard buffer does. The entire description is given below. This is a fairly comprehensive and conservative receptive model of a delay buffer. Because of the complex behavior, it is also somewhat lengthy. The net is a bit too complicated to easily understand from its graph, so we omit the diagram. We explain each part of the textual specification in turn since the description is fairly long.

```
(def (chaos-haz-buffer (a) (b) del)
   (wire-init-set! b (wire-init a))
```

We pass the delay specification in as a procedure to execute; this allows us to use the same description for different delay values. Also, we allow the initial value of the input to be anything, and we pass that initial value on to the output wire.

```
   (with-wires (chaos hazard)
      (complete ()
         (par
```

We need two internal wires. The first indicates whether we are in chaos mode: a positive transition indicates we are entering chaos mode, and a negative transition indicates that we are leaving chaos mode. Events on the second wire indicate the absorption of a single hazard by the buffer. The rest of the description is split into a set of concurrent parts.

```
(complete (hazard chaos)
   (forever a+ (del)
      (choice (b+) (hazard+ (ignore-timing))
                   (chaos+ (ignore-timing) chaos-))))
(complete (hazard chaos)
   (forever a- (del)
      (choice (b-) (hazard+ (ignore-timing))
                   (chaos+ (ignore-timing) chaos-))))
```

The first two parts are identical except that one handles rising edges and the other handles falling edges. In both, we allow an input event to occur, after which one of three things takes place. Either the output event occurs normally after some delay, or a hazard is absorbed by the gate (since each is complete on `hazard`, they both must cooperate to allow such a situation), or the model enters, and then later leaves, chaos mode.

```
(complete (chaos)
   (sm loop ((s 0))
      (case s
         (0 chaos+ (loop 1))
         (1 b+ (loop 1)) (1 b- (loop 1))
         (1 chaos- (loop 0))
         (1 chaos- :: b- (loop 0))
         (1 chaos- :: b+ (loop 0)))))
```

This next part describes one portion of chaos mode. We allow any output to occur while we are in chaos mode. When we leave chaos mode, we might have to cause an output transition; this is why we have three possible exit conditions.

```
(complete (chaos)
   (sm loop ((s 0))
      (case s
         (0 chaos+ :: a+ (loop 1))
         (0 chaos+ :: a- (loop 1))
         (1 a+ (ignore-timing) (loop 1))
         (1 a- (ignore-timing) (loop 1))
         (1 (del) chaos- (loop 0)))))
```

If we enter chaos mode, an input event must have caused us to enter. While we are in chaos mode, any input can occur at any time, and each one restarts the timer that allows us to leave chaos mode. When the timer expires, we signal an exit from chaos mode by lowering that wire.

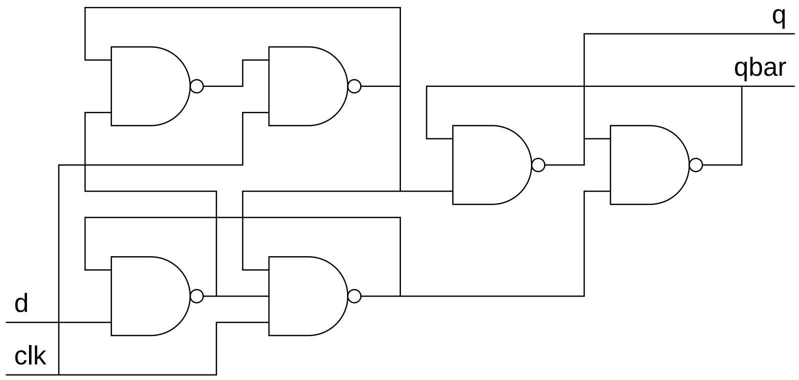


Figure 52. An edge-triggered flip-flop built out of six nand-gates.

```
(complete ()
   (sm loop ((av (wire-init a)) (bv (wire-init b)))
      (guard
         ((= av 0) a+ (loop 1 bv))
         ((= av 1) a- (loop 0 bv))
         ((= bv 0) b+ (loop av 1))
         ((= bv 1) b- (loop av 0))
         (1 chaos- (when (not (= av bv))
                      :: (if bv b- b+))
                   (loop av av))))))))
```

Finally, we track the values of the a and b wires so that when we exit chaos mode, we cause the appropriate output event.

The synchronization of all of the different components of the model is handled through use of `complete`. The complete model is implicitly receptive and has thirty transitions, fourteen normal places, eight epsilon places, and five orbits. There are only ten (untimed) reachable states. These are the four normal states of the fundamental buffer, two states where an output pulse (of the appropriate polarity) is pending, and four chaos states.

With this model, we analyze the operation of a common building block—the D flip-flop. This gate can be constructed out of six nand-gates as shown in figure 52 [48].

We specify the behavior of the untimed flip-flop as follows: we allow d to change at any time (since we are interested in the setup and hold times), but we define as a failure a rising edge on clk when there are pending output changes.

```
(def (ff (d clk) (q qbar))
   (wire-init-set! q (wire-init d))
   (wire-init-set! qbar (not (wire-init d)))
   (complete ()
      (sm loop ((dv (wire-init d)) (o (wire-init d)) (qv (wire-init d))
                (qbarv (not (wire-init d))))
         (guard
            ((= dv 0) d+ (loop 1 o qv qbarv))
            ((= dv 1) d- (loop 0 o qv qbarv))
            (1 clk+ (if (and (= qv o) (not (= qbarv o)))
                        (loop dv dv qv qbarv)
                        (void)))
            (1 clk- (loop dv o qv qbarv))
            ((not (= qv o)) (if qv q- q+) (loop dv o o qbarv))
            ((= qbarv o) (if qbarv qbar- qbar+) (loop dv o qv (not o)))))))
```

The timed implementation does not conform to the untimed specification because the latter allows the hold and setup times, as well as the cycle times, to be violated.

Let us first see what gate delays are allowed when we separate out the clock and data changes by a large margin. We start by giving each gate a delay of `:1,gatedelay:` and attaching the delay to the output of each gate. We use the following specification to space out the timing:

```
(def (timing (d clk) (q qbar))
   (process :inf: (choice (d+) (d-) (clk+) (clk-))))
```

If we specify ∞ as a lower and upper bound, we space the inputs to occur only when there are no other pending events; this is what is known as fundamental mode. When we verify, we find that any value for the gate delay will work.

If we use a more conservative approximation, attaching the delay elements to the input of each gate, the results are different; then, we find that the maximum gate delay must be less than twice the minimum gate delay. If we attach identical delay buffers to both input and output, then we find that the maximum delay must be less than three times the minimum delay. This shows that whether the delay is attributed to the gates or to the wires has a large impact on the results. The total model—with nineteen delay elements—has 718 transitions and 323 places—yet each verification takes under a second. However, our delay model was built to model a single gate delay, whether attached to the input or the output—so we will use the most conservative model, with the delays on the gate inputs, for the rest of our analysis. We also set the gate delay to `:1,1.99:`.

Our next investigation is of the required setup time for this model. We allow the input to change up to three times so that our buffers can go into chaos mode. A buffer in chaos mode can drive a following buffer into chaos mode, so three changes are sufficient. We use the following timing specification:

```
(def (timing (d clk) (q qbar))
   (process :inf: (choice (d+ d- d+) (d- d+ d-) ())
                  :setup,inf: clk+ :inf: clk-))
```

The system reports a required setup time of more than 4.97, which is three maximum gate delays less one minimum gate delay. If we only allow two transitions, so that the input does not go into chaos mode, we get the same value; by the time the gate goes into chaos mode, it no longer has any effect. Next, we consider the allowed hold time:

```
(def (timing (d clk) (q qbar))
   (process :100: clk+ :hold,inf: (choice (d+) (d-) ()) :100: clk-))
```

This yields a result of greater than 2.98, or two maximum gate delays less one minimum gate delay. Now we attempt to determine the delays from a clock change to the output. The timed places we introduce here are constraint places, so we may have as many as we like.

```
(def (timing (d clk) (q qbar))
   (par
     (process :inf: (choice (d+) (d-)) :4.98,inf: clk+ :inf: clk-)
     (process clk+ :0,qbardown: qbar- clk+ :0,qbarup: qbar+)
     (process clk+ :0,qup: q+ clk+ :0,qdown: q-)))
```

In order to see the resulting value, we have to use the `show-constraint-place-results` primitive.

`(show-constraint-place-results)` This primitive prints the observed values for all of the constraint places in the system.

The displayed output indicates that the rising edges occur between 2 and 3.98 units after the rising edge of the clock, so the delay to output is two gate delays for rising outputs. Falling edges occur between 3 and 5.97 units, so falling outputs have three gate delays.

With this style of investigation, we may easily determine other parameters such as minimum cycle time and required duty cycle.

6.10 Other Timed Petri Net Formalisms

Other techniques for adding timing to Petri nets have been widely used. We discuss some of the most common in this section and indicate why they are inappropriate for our purposes.

The original version of timed Petri nets associates timing with transitions [44]. The execution model fires a transition in two steps. The first step removes the tokens from the input places. Then, after the delay specified on the transition, the output tokens are placed in the output place. This type of timed Petri net is easily modeled by orbital nets (or, indeed, virtually any of the other types of timed Petri nets.) Unfortunately, these nets lack expressiveness. The only way to include timing is through the firing of a timed transition, but once the transition has started firing, there is no way to prevent it from completing if, for instance, a hazard occurs. More precisely, the resetting of a clock and the expiration of it are two separate events.

Perhaps the most popular way of adding timing to Petri nets is what is known as time Petri nets (without the "d") [34]. These nets also associate timing with transitions; the transition must be continuously enabled for the specified duration before it can fire. This form of Petri nets is extremely expressive (although it is not known if they can be modeled by orbital nets or if orbital nets can be modeled by them). Techniques for composing this type of net on shared actions, however, are not so obvious. For instance, a receptive gate might use many different transitions for a single input event; which transitions are enabled might change at any point because of other input events. Yet, we wish the time delay to be associated with the action, not the transition; these other input changes must not affect the timing of the specific action we are interested in. Essentially, the problem with these nets for our purposes is that timing is associated with the full enabling of the transition, so it is difficult to specify the timing in one part of the model without worrying about how other parts of the model affect that timing.

Another technique, taken from signal transition graphs, associates timing with the places [52], like we do. There are three primary differences between these techniques and orbital Petri nets. First, these techniques give a lot of attention to the different types of semantics possible when timing is associated with several different input places to a transition. Our formalism requires that there can be at most one such place. Second, there is no way to reset or vacate such a timed place before its timer makes it available; this restricts the expressiveness of the formalism. Third, there is no way to express constraints rather than behavior, primarily because these nets are defined as an executable formalism with no explicit failure semantics. These last two limitations required us to introduce our own formalism. It is an interesting topic for future research to extend orbital nets to allow multiple behavioral places into a single transition, and to decide which of the available operational semantics might be useful while still maintaining trace theory semantics.

The primary reasons that the semantics for orbital nets was chosen are the resulting simple constructions for composition, receptiveness, mirroring, and the conservative and liberal approximations. In the next chapter we see how much more efficient simulation and verification techniques than the ones presented in this chapter can be applied to the timing semantics of orbital nets.

Chapter 7

Timing II

... she was conscious of a comfortable feeling that the system, with all its inequalities and iniquities, would probably last her time. —'Saka' (H. H. Munro) 1870–1916

In the previous chapter, we presented a proof that discrete time is equivalent to continuous time for properties of our nets. We also provided a few examples of circuits analyzed using Orbits. The results we presented were accurate to four decimal digits, yet the largest verification time reported is just a few minutes. Discrete methods do not run this fast. Take, for instance, the Seitz queue example. In some cases, we have as many as seven pending events in the system. Each pending event has its own timer, and if we are using four significant digits, then each timer might take on about ten thousand values; obviously, a discrete state space of about 10^{28} states is beyond the capabilities of current machines. Even at one billion states per second, such a system would take over three billion centuries to investigate. Orbits requires only 2.0 seconds for this verification.

Consider a timed trace such as $[(a+, t_a), (b+, t_b), (c+, t_c)]$. To determine all possible states the system might attain using discrete approaches, we might have to consider all possible values of t_a, t_b, and t_c. In this chapter, we develop techniques to calculate, efficiently and in parallel, all states reachable by firing the above sequence for all possible values of t_a, t_b, and t_c These methods, called *geometric timing methods*, were originated by Lewis [28] and Berthomieu and Diaz [6]. We report improvements to existing geometric methods, including the reduction of the most common operation in such a verification from $O(n^3)$ to $O(n^2)$, and some techniques for determining the entire state space quickly. We also introduce an algorithm to compute all states reachable by all possible interleavings of concurrent events for a particular sequence. This last technique is especially fast and powerful.

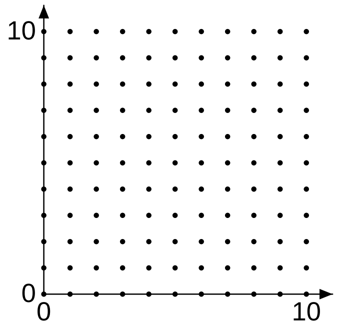


Figure 53. Timing component of state space using discrete methods.

7.1 Geometric Regions

Discrete timing methods enumerate every possible combination of integral clock values for each untimed state. Given two clock variables that can each take on values between zero and ten, we might have to enumerate 121 different combinations for each untimed state, as illustrated by figure 53. In general, if we have n clock variables, each of which has a maximum value of k, we might have $(k+1)^n$ timed states for each untimed state.

Not all formalisms for timed systems can be analyzed with discrete time. In Alur's dissertation, he proves that, in general, continuous time automata fitting fairly loose characteristics can be explored in a finite way using the unit cube approach. (The geometric region methods we discuss in this chapter also have this property.) This technique consists of splitting each clock value into an integer and a fractional portion. Then, equivalence classes of timed states are explored such that each equivalence class has the same integral parts of all clocks, and all fractional parts of clock values are in some specified order. This is possible because each state in an equivalence class has a single equivalence class as a successor when a transition fires or when time advances. A system with two clock variables might have the regions as shown in figure 54. Each triangle is a region, as is each intersection of lines and each line segment. Thus, we have two open triangles, one diagonal line segment, one single point, and one vertical and one horizontal line segment for each integer value less than ten, plus a few border regions; the total is 684 regions for the diagram pictured. In general, if we have n clock variables, each of which has a maximum value of k, we have approximately

$$\frac{n!}{(\ln 2)^n}$$

times as many regions as in the discrete case; for $n = 1, 2, 3, 4, 5, 6$, the exact values are $2, 6, 26, 150, 1082, 9366$.

Both the discrete case and Alur's method have the property that the size of the representation of each state and the calculations required for each state transition are linear in

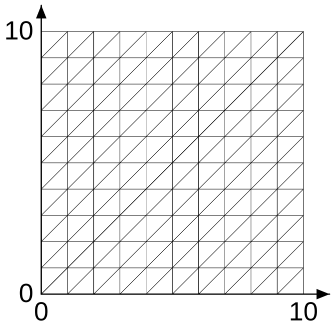


Figure 54. Timing component of state space using Alur's method.

the number of clocks. Unfortunately, these methods also have the property that the worst case behavior is common. In particular, as the limits on clock values increase, the state space also increases exponentially.

Alur's methods are geometric, in the sense that each equivalence class represents either one or an infinite number of continuous timed states. We would like to use a technique that can consider as many of these geometric regions as possible at once; that is, we might consider the entire square region in figure 54 as a single region.

Let us consider how to describe some reachable geometric set of timed states. The representation we develop is essentially identical to that in [16], except that we represent the ages of tokens (as time advances, they increase) while [16] represents the potential values of timers that decrease and expire as time advances.

We obviously need upper and lower bounds on clocks. In an n-dimensional space representing the potential clock values, these are $(n-1)$-dimensional planes, each perpendicular to one axis.

If we limit ourselves to this form of region, then we can describe the initial state of our nets, and we can describe the enabling conditions (which are just a conjunction of constraints of this form). But consider what happens when one transition starts a clock and separately enables another transition, and this latter transition also starts a clock. Since the second clock was started after the first, we must represent that the value of the second clock must be less than or equal to the value of the first clock. We cannot represent this using planes perpendicular to a single axis, but we can represent it by constraints on the difference between two clocks.

These two types of constraints are necessary and sufficient for our needs. More precisely, we represent a timed region by a set of constraints of the following forms, where c_i and c_j

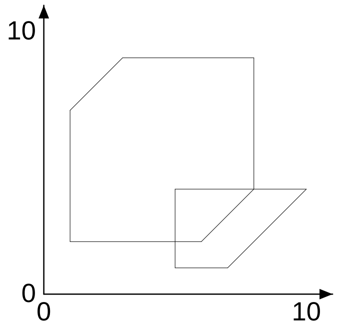


Figure 55. Timing component of state space using convex regions.

are clock variables, and k_i is an integer constant:

$$c_i \le k_i$$

$$k_i \le c_i$$

$$c_i - c_j \le k_i$$

We have put all of the inequalities in the correct order so that the only comparison we are using is $\le$. Since we do not allow $<$, $>$, or negation in our formalism, $\le$ will turn out to be sufficient. It is trivial to extend the ideas to allow $<$ and $>$. For instance, our square example is represented by the set of inequalities

$$\{0 \le c_i, c_i \le 10, 0 \le c_j, c_j \le 10\}.$$

Some other examples are given in figure 55. If we assume that c_i is the x axis and c_j is the y axis, the larger of the two regions is represented by the set of inequalities

$$\{1 \le c_i, c_i \le 8, 2 \le c_j, c_j \le 9, c_i - c_j \le 4, c_j - c_i \le 6\}.$$

The smaller region is represented by the set of inequalities

$$\{5 \le c_i, 1 \le c_j, c_j \le 4, c_i - c_j \le 6\}.$$

These regions are convex. A convex region is a region that, for any two points within the region, the midpoint of the two points is also within the region.

7.2 Operational Aspects of Geometric Regions

To use such a representation of regions for simulation and verification, we must represent the initial condition, calculate the enabling of transitions and the advancement of time, fire transitions, and compare two regions (to determine whether we have visited all of the states already). We also need to check that constraint places are satisfied.

These operations are performed on the system of inequalities that represents the current region of interest. The main operations are intersection (to add another inequality or combine two sets of inequalities), comparison, projection (to eliminate clock variables we no longer need), and an operation to advance time. We also need to find the maximum and minimum values for the various clocks. We will construct an efficient representation that supports all of these operations.

In such a system of linear inequalities, there can be redundant and underconstrained inequalities. For instance, of the three inequalities $x \leq 3$, $4 \leq y$, and $x - y \leq 7$, the third is redundant, since $x - y$ must be less than or equal to -1. The set of inequalities might also have no solution. The inequalities $x \leq 3$ and $5 \leq x$ are obviously inconsistent.

One possible format for storage maintains the inequalities in a minimal consistent set. That is, only those inequalities that are required are stored; no inequality whose omission would leave the convex region unchanged is stored. The example regions we gave before are in this form. Any convex polytope can be represented by such a set with one inequality for each plane that is not just a single point. Since each plane has a unique equation, and each convex region has a known number of planes, the solution is unique.

Unfortunately, with such a set, determining the minimum or maximum possible value of a particular clock is not simple. For instance, the set of inequalities for the smaller region in figure 55 does not contain an inequality of the form $c_i \leq k$; we have to determine the maximum value of c_i from the inequalities for the maximum value of c_j and the minimum difference of $c_i - c_j$.

So, we choose the opposite approach. We save all $n(n+1)$ possible inequalities for a particular convex region, such that each inequality is maximally constraining. That is, if any of the values on the right side of the $\leq$ are reduced, then the convex region would change. This means that if there exists any solution to the set of inequalities, then for each inequality, there is a solution with that inequality turned into an equation.

We also introduce a new variable c_0 that is always equal to zero. We use it to change inequalities of the form $c_i \leq k$ to $c_i - c_0 \leq k$, and inequalities of the form $k \leq c_i$ to $c_0 - c_i \leq -k$. With this change, all of our inequalities are in the same form.

Indeed, we may store just the constants in the inequalities by choosing a particular ordering, since we insist we have a full set. Thus, the set of inequalities for the larger region in figure 55 will be organized as follows:

$$\begin{pmatrix} c_0 - c_0 \leq \ \ 0 & c_1 - c_0 \leq 8 & c_2 - c_0 \leq 9 \\ c_0 - c_1 \leq -1 & c_1 - c_1 \leq 0 & c_2 - c_1 \leq 6 \\ c_0 - c_2 \leq -2 & c_1 - c_2 \leq 4 & c_2 - c_2 \leq 0 \end{pmatrix}.$$

We introduce zero diagonal entries for convenience. For any such matrix, only the constants will differ, so we store only the constants:

$$\begin{pmatrix} 0 & 8 & 9 \\ -1 & 0 & 6 \\ -2 & 4 & 0 \end{pmatrix}.$$

In this matrix, the maximum values of the clock variables are in the top row, and the minimum values are in the left column (but negated). For any inequalities that are not initially included, we set the corresponding matrix entry to ∞. We refer to this style of matrix as a *constraint matrix.*

The primary operation we perform on this system of inequalities is *canonicalization*—reduction of all of the inequalities to their most constrained value. Since the constants are always on the right side of the $\leq$, canonicalization corresponds to minimal constants. For a linear system of inequalities, a particular inequality is underconstrained, or the system has no solutions, if and only if there is some linear combination of the inequalities that has the same variable coefficients but tighter constants. Since the only coefficients in our inequalities are 1 and -1, this means that some set of inequalities must add up to a tighter inequality than the one in question. Assume we are concerned with the inequality $c_i - c_j \leq k$. Any such sum can be turned into a sequence p in the following way:

$$\begin{aligned} c_i - c_{p_0} &\leq k_0 \\ c_{p_0} - c_{p_1} &\leq k_1 \\ c_{p_1} - c_{p_2} &\leq k_2 \\ &\ldots \\ c_{p_m} - c_j &\leq k_{m+1}. \end{aligned}$$

such that the sum $k_0 + k_1 + \ldots + k_{m+1}$ is less than k.

Consider the matrix as an adjacency matrix for a weighted, directed graph, and the sequence p as a path through the graph. We are looking for the sequence p such that the sum of the weights along that path has the smallest value. This is precisely the all pairs shortest path algorithm, which can be solved by Floyd's algorithm in $O(n^3)$ time. That is, if we represent our system of inequalities as described above, then Floyd's algorithm is precisely our canonicalization algorithm. Floyd's algorithm has the advantage that it is particularly simple and elegant:

```
for (k=0; k<n; k++)
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         if (a[i,j] > a[i,k] + a[k,j]) then
            a[i,j] = a[i,k] + a[k,j]
```

Floyd's algorithm has the additional property that, when there is a loop with a total negative path weight, one of the diagonal entries will be set to a negative integer. If this happens, then the set of inequalities is inconsistent, and the constraint matrix represents a void region. (A region with a single point is not void.) However, the full generality of Floyd's algorithm is never needed; indeed, for all of the steps needed for timing verification using orbital nets or many other varieties of timed automata, specializations of Floyd's algorithm that run in time proportional to n^2 are sufficient. We will discuss these improvements in a later section.

Timing verification proceeds precisely as does untimed verification. We wish to prove a property of the full set of traces generated by a model. The model has a finite amount of state, and this state completely determines future behavior. For safety properties, which check that no bad state is ever reached, we simply need to explore and check every reachable state.

The set of reachable states forms a (usually cyclic) *reachability graph*; the states are the nodes and they are connected with state transitions. Since every reachable state must either be the initial state or have a predecessor state, computing every successor to every reachable state is guaranteed to find the entire state space. This *finite-state verification procedure* allows us to reason about an infinite set of infinite sequences in a finite amount of time.

Time is introduced as another component of state; for orbital nets, time is represented by the ages of all timed tokens. Our proof of equivalence between discrete and continuous time allows us to consider only a finite number of token ages, and thus a finite state space. This chapter improves things further, by taking advantage of certain properties of time to consider a large collection of timed states and state transitions in parallel. With geometric methods, each node in the reachability graph consists of an untimed state and a timed state region representing many timed states. Given a timed state region and a particular transition to fire, we can calculate the timed state region that represents all timed states reachable by firing that transition or advancing time from any timed state in the original region. Thus, we are simply using constraint matrices as a representation of an aggregate of timed states; in all other respects, verification proceeds as before.

Let us presume that we are performing verification on a timed system. We have just encountered a new timed region for an untimed state and we must calculate the successor regions. We assume that the current region matrix is canonical and that the upper limits of all of the clock variables have been combined into the timed region, so no clock regions exceed their maximum at this point.

The operations we perform are:

1. Determine which transitions are timing-enabled in the current timed region.
2. Select one transition to fire.
3. Calculate a new timing region, restricted to that subset of the previous timed region in which the transition we just selected was enabled. Canonicalize the resulting matrix.
4. Examine the constraint timing variables associated with input places to this transition, and determine if firing this transition will cause any of them to violate their minimum timing age. Report the failure, if so.
5. Remove the timing variables that are consumed by the firing of this transition by projection.
6. Introduce any new timing variables created by the firing of this transition, maintaining their relation to the existing timing variables. Canonicalize the resulting matrix.
7. Allow time to advance as much as allowed by the maximum ages of tokens in the behavioral places, and canonicalize the resulting matrix. Determine if this time advancement might cause any constraint places to exceed their maximum age; report a failure, if so.
8. Normalize the timing variables that have no explicit maximum; this step keeps the state space finite.
9. Determine whether this timed region has been seen before. If it has, backtrack; otherwise, add it to the hash table.
10. Calculate the new set of untimed-enabled transitions for this set. Make sure that each marked behavioral place has an untimed-enabled transition that consumes the token. Repeat from the start.

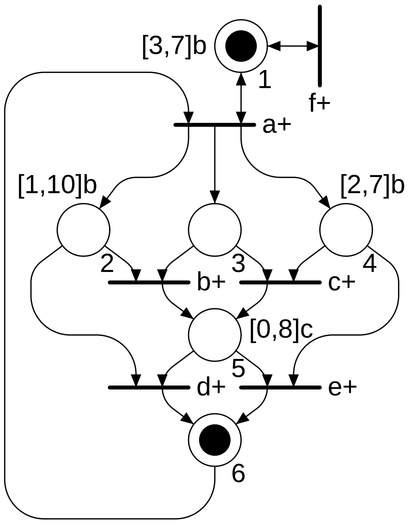


Figure 56. Execution example.

A difference between this sequence of operations and the one we described in the previous chapter is that, in the sequence just introduced, we advance time after a transition firing. We want each timed region to represent as many states as possible, and new states can be reached by advancing time. This is strictly a perspective shift and makes no behavioral change.

To illustrate our successor calculation, we shall use the example in figure 56, hereafter referred to as the execution example. In this figure, we have numbered the places one through six. The four reachable untimed states, represented as sets of places, are $\{1, 6\}$, $\{1, 2, 3, 4\}$, $\{1, 4, 5\}$, and $\{1, 2, 5\}$. We shall start at the point in depth-first search where we have fired the $a+$ transition; at this point, we are in the untimed state $\{1, 2, 3, 4\}$ and our canonical region matrix is

$$\begin{pmatrix} 0 & 7 & 7 & 7 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where row two corresponds to place one, row three corresponds to place two, and row four corresponds to place four. This matrix represents a portion of the line $x = y = z$ in three-dimensional space, indicating that all timed tokens have the same age. Since the firing of $a+$ created each timed token, this is reasonable. In addition, the minimum age is zero and the maximum age is seven.

Now we consider each step in turn. The first operation is to determine which transitions are enabled. We start by determining which transitions are untimed-enabled, ignoring the timing conditions; the resulting set consists of those transitions labeled $\{b+, c+, f+\}$. Next, we eliminate from this list any transition that is not enabled in any timed state within our geometric region. A transition that is untimed-enabled, and either has no input behavioral places that are not ignored, or whose behavioral place (if any) contains a token with an age in the range allowed by that place, is timed-enabled. Our timing semantics never allows the age of a token in a behavioral place to exceed the maximum allowed age, so we only need check the minimums. Since we only allow a maximum of one non-ignored behavioral place in a transition, the timed-enabling check reduces to a simple comparison of entry $a[0, i]$ of our current timed region with the minimum of the behavioral range, where i is the matrix row and column number corresponding to the appropriate behavioral place. If the maximum clock value indicated by the timing matrix is greater than or equal to the minimum in the behavioral range, then that transition is timing enabled. In this case, since all tokens can reach age seven, and since this is greater than the minimum age for all three behavioral places, all three transitions are timed-enabled.

In step two, one enabled transition is chosen to generate the next state. Depth-first search requires us to consider all transitions eventually; we choose the transition labeled $b+$ and push the current state and the remaining transitions onto a stack for later consideration.

Step three restricts the current region to that subset for which the transition is enabled. Since we only allow a single behavioral place, this operation introduces the inequality that requires the minimum age of the token in the behavioral place to be greater than or equal to the minimum firing age allowed by that place. If the transition has no input behavioral place that is not ignored or if the condition is satisfied for each timed state in the region, we skip this step. Otherwise, we introduce this additional constraint by reducing the appropriate single value in column one of the matrix, and then recanonicalizing it. Since we have only reduced a single value in the matrix, and the matrix was originally canonical, we use a faster $O(n^2)$ algorithm to recanonicalize. In our example, the $b+$ firing consumes the timed token from place two, which indicates a minimum firing age of one. The third row and column correspond to this token, so we reduce entry $a[2, 0]$ to -1, yielding

$$\begin{pmatrix} 0 & 7 & 7 & 7 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Then we recanonicalize, obtaining

$$\begin{pmatrix} 0 & 7 & 7 & 7 \\ -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}.$$

The resulting matrix includes only those seven discrete states in which $b+$ was actually enabled to fire.

In step four, we need to make sure that firing a transition does not cause a token from a constraint place to violate its minimum age requirement. For any canonical matrix, we know that for each inequality, there exists a solution of the set of inequalities for which the left side of that inequality is equal to the right side. Thus, if the minimum value allowed by the constraint matrix for any constraint place consumed by this transition is less than the minimum age requirement of that constraint place, we can indicate an error immediately; this involves simple inspection of the first column of the matrix. Since we have no currently marked constraint places in our example, we do nothing.

Step five is projection; we perform this by eliminating the rows and columns of the matrix that store values for clock variables being consumed. In this case, we eliminate row and column three of the matrix; the result is

$$\begin{pmatrix} 0 & 7 & 7 \\ -1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}.$$

If a matrix is canonical, any projection of that matrix is also canonical. We prove this by observing the definition of a canonical constraint matrix—all inequalities are maximally constrained. There is no linear combination of the other inequalities that yields a more constrained matrix. If this was true in the unprojected matrix, it must be true in the projected matrix, as we have simply reduced the set of inequalities.

This step requires a complete set of inequalities; if we have only a partial set, then the projection operation becomes more complex. Consider the set of inequalities

$$\{5 \leq c_i, 1 \leq c_j, c_j \leq 4, c_i - c_j \leq 6\}.$$

We cannot project this system of inequalities onto the c_i axis by eliminating all inequalities that contain other variables; we would be left with no maximum value for c_i. Yet, c_i clearly has a maximum value, since $(c_i - c_j \leq 6) + (c_j \leq 4) = (c_i \leq 10)$.

In step six, we introduce new timing variables. According to our execution semantics, any new tokens must be initialized with an age of zero, so the minimum and maximum bounds are set to zero. The maximum difference between an existing clock variable and any new clock variable is the maximum value of the old clock variable; this difference will remain constant as time advances. In addition, the minimum difference is the minimum value of the old clock variable. Finally, the minimum and maximum difference between clock variables introduced by the same transition firing is always zero. So if our original matrix was

$$\begin{pmatrix} 0 & (\text{old clock maximums}) \\ \begin{pmatrix} \text{old} \\ \text{clock} \\ \text{minimums} \end{pmatrix} & \begin{pmatrix} \text{old clock} \\ \text{difference} \\ \text{constraints} \end{pmatrix} \end{pmatrix}.$$

then, when we introduce some new clock variables, the resulting matrix is

$$\begin{pmatrix} 0 & (\text{old clock maximums}) & 0 \\ \begin{pmatrix} \text{old} \\ \text{clock} \\ \text{minimums} \end{pmatrix} & \begin{pmatrix} \text{old clock} \\ \text{difference} \\ \text{constraints} \end{pmatrix} & \begin{pmatrix} \text{old} \\ \text{clock} \\ \text{minimums} \end{pmatrix} \\ 0 & (\text{old clock maximums}) & 0 \end{pmatrix}.$$

The resulting matrix is obviously canonical because each introduced row and column is a copy of some original row and column zero. For our example, firing of $b+$ introduces a timed token in place five. The resulting matrix is

$$\begin{pmatrix} 0 & 7 & 7 & 0 \\ -1 & 0 & 0 & -1 \\ -1 & 0 & 0 & -1 \\ 0 & 7 & 7 & 0 \end{pmatrix}.$$

This represents the set of states reachable upon firing of $b+$ without advancing time. There is a total of seven discrete timed states represented by this matrix.

Step seven models the advancement of time. Advancing time only increases clock variables, so the minimum values remain the same (we allow the advancement of time, but we do not require it). Since all of the clocks advance at the same rate, the difference constraints among the clock variables remain fixed. Only the maximums of the clock variables change.

The sole restriction on how much a clock can advance is the limit on how much global time itself can advance. Our semantics defines that time can only advance such that no behavioral place token exceeds its maximum age. This requirement can be expressed as a convex region consisting of the conjunction of the inequalities $age \leq max$ for each behavioral token. Thus, to advance time, we remove the maximum age constraints in the current timing region; these were the maximum ages of the tokens at the point when the current untimed state was entered, and we calculate the full set of reachable states including those encountered by advancing time. We replace the existing values with ∞. We then take the intersection of the resulting region with the region created by restricting the maximum values of the clocks corresponding to the behavioral places; this limits the advance of time to *max-advance*. This intersection operation involves taking the union of the sets of inequalities, removing the underconstrained ones, and then recanonicalizing the matrix. Geometrically, we extrude the shape along the $c_1 = c_2 = \ldots = c_n$ axis to the maximum allowed by the behavioral places. For our example, we replace the maximum values in the constraint matrix for the tokens in places one and four with their respective maximum allowed ages, and we replace the maximum value for place five with ∞. The resulting matrix is:

$$\begin{pmatrix} 0 & 10 & 7 & \infty \\ -1 & 0 & 0 & -1 \\ -1 & 0 & 0 & -1 \\ 0 & 7 & 7 & 0 \end{pmatrix}.$$

Recanonicalizing this matrix yields

$$\begin{pmatrix} 0 & 7 & 7 & 6 \\ -1 & 0 & 0 & -1 \\ -1 & 0 & 0 & -1 \\ 0 & 7 & 7 & 0 \end{pmatrix}.$$

This matrix represents the twenty-eight discrete timed states reachable by firing $b+$ and then advancing time from the original timed region.

After advancing time, we examine the constraint place variables in the resulting timing region and determine if any of the maximum values are exceeded. If so, we report a failure immediately. In this case, the maximum age of six for constraint place five is less than eight, so no error is reported.

Step eight, normalization, is more complex, and we will describe it in its own section below. Normalization insures that our matrix entries remain bounded, even in the presence of infinite upper limits on timed places.

Step nine determines whether we have seen this set of timed states before. If we have not, we add it to our table of seen states, and move on to step ten. If we have, we can backtrack and explore some other pending state and transition from our stack; if the stack is empty, we are finished with the verification.

In step ten, we calculate the new set of untimed-enabled transitions for the current region. In our case, this is the set of transitions labeled $\{c+, f+\}$. We then make sure that each marked behavioral timed place can have its token consumed; indeed, $c+$ consumes the token from place four, and $f+$ consumes the token from place one, so this requirement is satisfied.

We now repeat at step one, with the new untimed state and constraint matrix.

7.3 Normalization

To illustrate normalization, which is step eight in the preceding section, we will use the example shown in figure 57. Normalization handles behavioral or constraint places that have no upper limit. Since the clock variables are unbounded, they might take on any

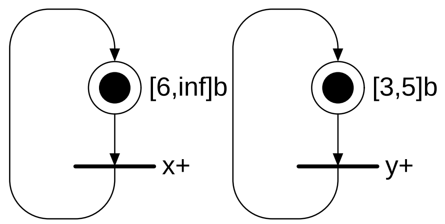


Figure 57. An example timed net illustrating why normalization is important.

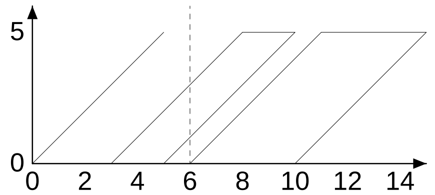


Figure 58. Three reachable timed state regions of the normalization example.

nonnegative value, and thus the reachable state space of timed regions would be infinite. On the other hand, once the value of the clock exceeds its minimum value, the value no longer matters; we only need to remember that the minimum was exceeded. In the discrete case, we handled this by setting any such clock that exceeded its minimum back to its minimum plus one, which resulted in a finite state space. With our more complex geometric approach, we need a different technique. The process of normalization ensures that no entry in a constraint matrix is unbounded.

Consider, for example, the simple net shown in figure 57. The initial region and the two regions reached by firing $y+$ are shown, left to right, in figure 58, where the horizontal axis is the clock bounded by $[6,\infty]$ and the vertical axis is the clock bounded by $[3,5]$. The dashed line is at six, because any values for the horizontal clock greater than six are equivalent to six.

The initial region has a constraint matrix of

$$\begin{pmatrix} 0 & 5 & 5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix};$$

after the nth firing of the transition labeled $y+$, the constraint matrix is

$$\begin{pmatrix} 0 & 5n+5 & 5 \\ -3n & 0 & -3n \\ 0 & 5n & 0 \end{pmatrix}.$$

If we continue firing $y+$, then we will get larger and larger regions that are closer to infinity in the horizontal direction; the constants in the constraint matrices will be unbounded.

To define a technique that deals with this problem, we need to define precisely what sets of timed states are considered equivalent. We will use this definition of equivalence classes of states to reduce the size of some timed regions, such that they still contain a representative timed state in every equivalence class represented by the original timed region. This reduction operation will always return a constraint matrix whose constraints satisfy finite upper bounds. Since our constants are always integers, a finite state space of timed state regions is produced.

The equivalence class relation is defined as follows: two states, s and s', with vectors v and v' representing the respective ages of the timed tokens in some well-defined order, belong to the same equivalence class if and only if $f(v) = f(v')$. The primary idea behind f is that it is a canonical and unique member of an equivalence class of timed states; f is defined as $f(v)_i = min(v_i, premax_i)$, where *premax* is the function that indicates the maximum age of a token beyond which all ages are equivalent.

If the timing requirement of a place is given as $(a, b, type)$, then, once a token's age exceeds b, its age is moot. (If *type*=*behavior*, our execution semantics will never allow the age to exceed b; if *type*=*constraint*, it can exceed b, but the firing will be a failing firing.) We define the value of *premax* for a token in such a place to be $b + 1$. If it were just b, we would not notice when a constraint place exceeded its maximum age. Similarly, for a timing requirement given as (a, ∞), once a token's age exceeds or equals a, its age becomes moot. We define *premax* for a token in these places to be a. Thus, *premax* is a function of the place in which the token resides.

The age of a token is only used in comparing it with the constraints on the timed place and thus defining the successive reachable timed states within one untimed state through the advancement of time. Any satisfactory equivalence class relation between states must have identical timed properties, identical enabled transitions, and equivalent reachable states. The equivalence class we define trivially satisfies the first and second conditions. The third condition is satisfied by considering the advancement of time and the firing of two transitions separately.

Time advances by incrementing each token age by the same amount. Consider two equivalent timed states, s and s'. Partition the tokens in each state into two sets: one with ages less than the corresponding *premax*, and one with ages greater than or equal to *premax*. If the states are equivalent, this partition will be identical for the two states. Those tokens in the first set must have equal ages in order for the states to be equivalent, so after advancing time, they will again have equal ages. Those tokens in the second set are all greater than their corresponding *premax*, and advancing time will not change this condition; therefore, after time advances they will still map to the same canonical value.

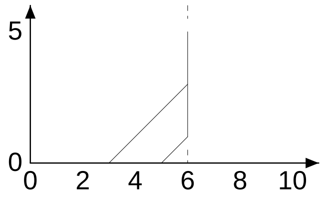


Figure 59. One normalization technique.

Therefore, advancing time from two states that are equivalent will always result in two equivalent states.

Firing transitions also causes two equivalent states to map to equivalent successor states. New tokens are given an age of zero, so their canonical representative also has an age of zero. Any preserved old tokens retain their old values; if they were originally equivalent, they will still be equivalent after the transition firing.

Thus, the equivalence relation is satisfactory. If, in any given timed state we have several members of a given equivalence class, we may discard any proper subset of this equivalence class without changing the behavior or properties of the net.

Consider what this means for the middle timed region of figure 58, with *premax* for the horizontal token equal to six. All timed states at the same vertical height within the region to the right of the dashed line are in the same equivalence class. As we did in the discrete case, we may replace each equivalence class by its canonical representative, yielding the timed state shown in figure 59. The entire region to the right of the dashed line has been replaced by a single vertical line.

This approach would work fine, except that the resulting region is no longer convex. We could separate the resulting region into two separate convex regions and explore each separately. In general, it might be necessary to split a region with n clocks into n separate regions in order to maintain convexity. Since each region has a representation of size $O(n^2)$, the total complexity would be $O(n^3)$. This technique violates the useful property that a given timed region leads to a single successor timed region when a specific transition is fired. So, although such a technique would work, we instead use a more complicated but faster technique that preserves the $O(n^2)$ worst-case complexity.

The technique we introduce is based on finding, for each clock that might exceed its corresponding *premax*, a new maximum value that is as small as possible without changing the set of equivalence classes of timed states represented by the timed region. Consider once again the middle timed region in figure 59. The minimum maximum for the horizontal clock that does not change the set of equivalence classes is shown in figure 60. If this new maximum were any more to the left, the equivalence class with the vertical clock having a

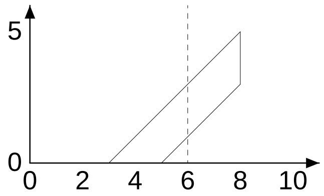


Figure 60. A better normalization technique.

value of five would not be represented. In addition, the shown region has every equivalence class in the original region, so no greater maximum would be the minimum maximum.

This alone is not sufficient, however. Consider the trend of the timed regions as shown in the original figure; the minimum of each timed region is to the right of the minimum of the previous timed region. Indeed, if we use normalization as we have presented it so far, the constraint matrix after the nth firing of $y+$, for $n > 0$, would be

$$\begin{pmatrix} 0 & 3n+5 & 5 \\ -3n & 0 & -3n \\ 0 & 3n+2 & 0 \end{pmatrix},$$

so it is still unbounded. All but the first two regions have a horizontal minimum greater than *premax*, and this observation leads to a solution. When the minimum of a given clock is greater than *premax*, we introduce a simple change of variables to make the minimum equal to *premax* without changing the equivalence classes represented by the region. If the minimum is m, then the change of variables we introduce is $(v_i)' = v_i - m + premax_i$. This change of variables is done by adding $m - premax_i$ to each entry of column i and by subtracting this value from each entry of row i in our original constraint matrix. This prenormalization step has a total worst-case complexity of $O(n^2)$. Typically, it is not needed often, and when it is, only a single clock variable needs to be adjusted; so, in practice it does not add much to the total run time. The code is given below:

```
for (i=1; i<n; i++)
   if (a[i,0] + premax[i] < 0) then
      delta = a[i,0] + premax[i]
      for (j=0; j<n; j++)
         a[i,j] = a[i,j] - delta
         a[j,i] = a[j,i] + delta
```

After performing this step, we proceed with our main normalization.

Determining the minimum maximum that we may use for a clock is straightforward. Using the values in the *premax* vector might constrain the reachable values of some other clock to fall below its *premax* or its current maximum. because of the constraints on the differences between clock variables. Thus, we choose the minimum value for each clock value that does not restrict the other clocks in this fashion. And, of course, we only want to reduce those clocks whose maximum already exceeds *premax*. More precisely, for each i, we need to set

$$a_{0i} = \underset{j}{MAX}\left(min(a_{0j}, premax_j) - a_{ij}\right).$$

This is the *normalization invariant.* The following algorithm performs this in $O(n^2)$ time:

```
fixup = ()
for i in 1..n
   if a[0,i] > premax[i] then
      t = premax[i]
      for j in 1..n
         if min(a[0,j], premax[j]) > t + a[i,j]
            t = min(a[0,j], premax[j]) - a[i,j]
      if t < a[0,i] then
         fixup = i :: fixup
         a[0,i] = t
```

This algorithm creates a list of matrix entries that have been reduced, called *fixup*. For each i not in *fixup*, the maximum entries remain the same. If i is in *fixup*, then both the original and the new values of a_{0i} are greater than or equal to $premax_i$. Thus, the expression $min(a_{0j}, premax_j)$ is the same for the original and the modified matrix for each j. This shows that each iteration of i does not affect the other iterations, even though the maximum values are changed in the array.

We then recanonicalize the array with the simple $O(n^2)$ loop:

```
if fixup
   for i = 1..n
      for j in fixup
         if a[i,0] + a[0,j] < a[i,j] then
            a[i,j] = a[i,0] + a[0,j]
```

The normalization invariant is still true, as we will now show. Entry a_{ij} is only reduced if j is in *fixup*; it is reduced to $a_{i0} + a_{0j}$. Since for a canonical matrix, $a_{0i} + a_{i0} \geq 0$, and since $a_{0j} \geq premax_j$, we add these two inequalities to get $a_{0i} \geq premax_j - (a_{i0} + a_{0j})$. Thus, the revised $min(a_{0j}, premax_j) - a_{ij}$ values are still smaller than the originally calculated a_{0i}, so the normalization invariant must hold.

In practice, the maximum for a particular clock does not exceed its *premax* value often, and when it does, usually only a small number of clocks do so at the same time. Because of this, the above two sections of code typically do not significantly influence the run time of the overall timed step.

We prove that this normalization process does not change the set of equivalence classes.

Theorem 7.1. A matrix a represents the same set of equivalence classes of timed states as the matrix a' that results from the normalization procedure.

Proof. Every equivalence class in a' is in a, since a' is a subset of a. To show that each equivalence class in a is in a', we assume there exists some timed state s in a no member of whose equivalence class is in a', and we show that this leads to a contradiction.

Let v represent the timing ages of s. Create a new v' by replacing the elements of v with $min(v_i, a'_{0i})$. This new state is in the same equivalence class, since either $a'_{0i} = a_{0i}$ or $a_{0i} \geq a'_{0i} \geq premax_i$. We show that v' is allowed by a', which will prove that it is also allowed by a.

Because of the way it is constructed, v' meets the maximum clock constraints imposed by a'. Since a' is canonical, $a'_{0i} \geq -a'_{i0}$, so it also meets the minimum constraints. Let us assume that v' violates some constraint a'_{ij} because $v'_j - v'_i > a'_{ij}$.

If j is not in *fixup*, then $a_{ij} = a'_{ij}$ and $a_{0j} = a'_{0j}$, so $v'_j = v_j$. In order for the inequality to have been violated, $v'_i > v_i$. Yet, $min(v_i, a'_{0i}) \leq v_i$, so by contradiction, j must be in *fixup*.

If $a_{ij} \neq a'_{ij}$, then $a'_{ij} = a'_{i0} + a'_{0j}$. But $v'_j = min(v_j, a'_{0j}) \leq a'_{0j}$, and $v'_i = min(v_i, a'_{0i}) \geq -a'_{i0}$. We have already shown that the minimum and maximum values must be satisfied, so by taking the difference between the previous inequalities, $v'_j - v'_i \leq a'_{0j} + a'_{i0} \leq a'_{ij}$, which contradicts our assumption that there is a violation. Therefore, in order for there to be a violation, a_{ij} must equal a'_{ij}, with j in *fixup*.

If $a_{ij} = a'_{ij}$ and j is in *fixup*, then $v'_i < v_i$, so i must also be in *fixup*, and $v_i > a'_{0i}$. Yet, $v'_j \leq a'_{0j}$ and $v'_i = a'_{0i}$, so $v'_j - v'_i \leq a'_{0j} - a'_{0i}$. Since the matrix is canonical, $a'_{0i} + a'_{ij} \geq a'_{0j}$, so by addition of inequalities, $v'_j - v'_i \leq a'_{ij}$, which indicates that there is no violation. Therefore, by contradiction, the two matrices represent the same set of equivalence classes. ∎

Finally, we show that the matrix entries themselves are bounded. Under no conditions can the age of a clock go negative, and prenormalization forces every minimum to be less than or equal to the corresponding *premax*, so

$$-premax_i \leq a_{i0} \leq 0.$$

For a canonical matrix, $a_{i0} \leq a_{ij} + a_{j0}$, so

$$a_{ij} \geq -premax_i.$$

Because our normalization invariant must always hold, for every i there exists some j such that $a_{0i} = min(a_{0j}, premax_j) - aij$. This value can be no larger than $premax_j + premax_i$. The maximum value of a clock can also never be negative, so

$$0 \leq a_{0i} \leq premax_i + \underset{j}{MAX}\, premax_j.$$

Because $a_{ij} \leq a_{i0} + a_{0j}$, we also set an upper limit on the internal entries of the matrix:

$$a_{ij} \leq premax_j + \underset{k}{MAX}\, premax_k.$$

As an illustration of normalization, consider the case where all timed behavioral places in an orbital net have timing requirements of $[0, \infty]$. In this case, *premax* will be equal to zero for all of these places, and by the above bounds, all matrix entries in a canonical, normalized, timed constraint matrix must be zero. This is not the case without normalization, even when normalization is not required to keep the matrix entries bounded. It also shows that for such a system, the timed reachability graph is precisely the same as that for the untimed case.

7.4 Complexity of Geometric Region Verification

The worst-case complexity of the timed region state space exploration process is dependent on the magnitude of the parameters if they do not have a common factor. While the technique does perform extremely well for most examples, the worst-case behavior of this technique is worse than either the discrete case or the unit cube case. As we shall see, we seldom encounter the worst-case behavior in practice.

Our results in the prior chapter are still useful. The equivalence between discrete and continuous time for orbital nets can be used with geometric methods. For instance, some liveness and state graph minimization techniques, as well as one of our improvements to the conventional geometric region method, require negating a constraint or taking the difference between two timed regions. Without the equivalence between discrete and continuous time, our regions would have to support the $<$ operation as well as $\leq$, since $\neg(a \leq b) \equiv (b < a)$. This would complicate every operation on timed regions. We instead use the simpler $\neg(a \leq b) \equiv (b \leq a - 1)$, which is true on integer values.

Let us consider the simplest use of these timed regions for verification. We consider each timed region as a separate node in our reachability graph; the edges are transition firings. This is the most commonly used technique, as it provides an exact characterization of the reachability graph for the continuous state space.

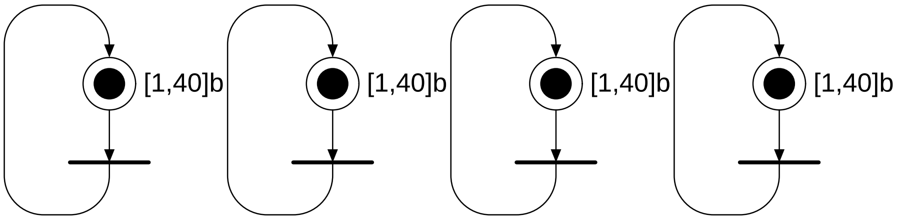


Figure 61. The adverse example, for $n = 4$ and $k = 40$.

When we use this technique on our execution example presented in figure 56, we obtain a reachability graph with twenty timed regions and thirty-six edges. Some discrete timed states are represented in several timed regions, but each possible discrete timed state is represented in at least one timed region.

Another example, provided by Alur in his dissertation, is a simple three-element concurrent system modeling a train, traffic gate, and controller. Using this technique, we find that the entire state space is represented by just sixteen of these timed regions, with forty-three edges between the regions; the run time is less than a quarter of a second. For comparison, using the unit cube approach, eight thousand nodes and fifty thousand edges are required to represent the reachable state space.

Similarly, when we analyze the Seitz queue element (shown in the previous chapter) using discrete methods, we find a state space of 8,753 states when the timing values used are five and six; using slightly more accurate timing values generates explosively more states. Using the timed region approach generates just 3,236 regions, for any timing parameters for which the circuit works, no matter what their magnitude.

The exact region approach is not always an improvement. For the net shown in figure 61, the exact method generates 219,977,777 distinct regions (asymptotically $1.6k^5$). We refer to this net as the adverse example. The discrete case would generate 2,825,761 regions (asymptotically, k^4); the unit cube case would generate at least 122,880,000 regions (but asymptotically, $48k^4$). This example was specifically constructed to illustrate the possibly poor performance of the timed region method. While the timed method does not appear much worse than the unit cube method, as n and k increase, the difference becomes more pronounced. In general, the worst-case performance for the exact region method is probably somewhere between

$$|S|(k/2)^{2n}$$

and

$$|S|(2k)^{n^2}$$

The lower limit is based on an approximation for the state space of the above example; the upper limit is based on the total number of possible regions expressible by a timed region matrix. Even the lower limit is much worse than the unit cube or discrete method.

In the next several sections, we will show some improvements to this method that can make it even more effective.

7.5 Subset Verification

The first technique is based on the geometric observation that many regions might be subsets of previously seen regions. If, during a depth-first search of the state graph, a new timed region is found to be a subset of a previous timed region, then every timed state represented in the new region has already been or currently is being explored; so, we immediately backtrack. Further, if we encounter a region that is a direct superset of a previous region, and all of the successors to that previous region have not been calculated, we need not explore them since we will find them all from the current, larger region.

The drawback of this technique is that we must compare any newly reached region with every other timed region that has been found for that untimed state. Thus, we no longer can hash the state space to quickly determine if the state has been seen before. Instead, we execute a linear search, comparing the new region with all old regions found for that untimed state and check to see if this region is either new, a superset, or a subset of previously seen regions. In practice, the number of regions for each untimed state tends to be small at any given time. Even if the number of untimed states is large, the elimination of many timed regions from future exploration saves time.

The subset calculation is fast. Specifically, one timed state region is a subset of another if every constant in the matrix representing the first is less than the corresponding entry in the other. Thus, the comparison of two regions is linear in the size of the representation.

While the set of reachable timed states is exactly the same as in the previous method, the set of edges connecting the regions is no longer exact. In general, each edge we find represents a superset of the exact edges.

This technique's worst-case performance may be worse than that of the exact region method because of the linear scan for subsets in a particular untimed state. We have not been able to find a lower limit for the worst-case performance than that of the exact method. Nonetheless, the subset method improves the performance for all of the examples we have tried.

Consider our execution example again. The initial constraint matrix for our execution example, before any transition firings, is

$$\begin{pmatrix} 0 & 7 \\ 0 & 0 \end{pmatrix}.$$

This constraint matrix represents eight discrete states for the possible eight discrete ages of the token in place one. After executing the trace $[a+, c+, d+]$, we are in the same untimed state, but our constraint matrix is now

$$\begin{pmatrix} 0 & 7 \\ -5 & 0 \end{pmatrix}.$$

This matrix represents three discrete untimed states, each of which is a member of the original constraint matrix. Thus, we have seen no new states, and we can backtrack immediately. Using the subset method reduces our reachability graph for the execution example to eleven timed regions and twenty-four edges.

This technique makes a dramatic improvement in run time for our adverse example. Without using the subset method, setting k to ten and n to four in order to get a reasonable run time finds 440,645 timed regions and 1,755,136 edges using 1,941 seconds of CPU time. For $n > 4$, the computer (with 64 megabytes of memory) runs out of memory. When we use the subset method, we find twenty-four timed regions and 1,367 edges using 4.3 seconds of CPU time. This is a run-time improvement of about 450.

The reason we see twenty-four final regions using the subset method is because, with four clock variables, there are 4! or twenty-four possible orderings of the transition firings that last reset the clocks. Indeed, this will be the case in general using the subset approach; if there are n clocks in an untimed state, and there are no restrictions on the order in which the appropriate transitions occurred, then there will usually be $n!$ timed regions. In some cases there will be fewer, and in some cases there will be more.

If we try and push the subset method a little harder by setting k to ten and n to five, the program still completes; it finds 120 timed regions and 23,622 edges using 605 seconds of CPU time. The next size, with an n of six, does not complete in eight hours of CPU time.

7.6 Coalescing Regions

The previous method is based on the observation that, if one region is a superset of another, the smaller region need not be remembered or explored. It passively finds regions, relying only on transition firings. The next approach attempts to construct larger regions by combining smaller ones, in the hope that the reachable regions from a larger region will be larger than the reachable regions from a small region.

As each new region is found, rather than checking for set inclusion with previous regions, we attempt to combine it with one or more of the existing regions to construct a larger region. Every point in this larger region must be a member of at least one of the two regions being combined.

Unfortunately, the union of two convex regions might not be convex. For instance, the union of the two figures in figure 55 is not convex. Determining whether the union of two convex regions is convex, even given the restricted structure of our convex regions, is not a trivial operation, especially when each such convex region can have $n(n+1)$ planes and an exponential number of vertices.

If the union is indeed convex, then finding the union is as easy as finding the intersection; for the latter, we take the minimum, or tighter, values of corresponding matrix entries; for the former, we take the maximum, or looser, values. Unlike the case for intersection, if the union is convex, then the simple construction mentioned before yields a matrix that is already canonical.

We now present an algorithm that determines if the union of two convex regions, represented as constraint matrices, is convex. If so, we will calculate the matrix.

Our algorithm is based on the observation that if the union of two convex regions is not convex, then there must be a point in the convex hull formed by the union of the two regions that is not in either region. The convex hull of a set of points is the smallest convex polygon enclosing all of the points. If a point is not in a convex region represented by a constraint matrix, then that point does not satisfy some inequality in that region. We attempt to identify one constraint from each of the two regions being joined such that some point in the convex hull of the two regions violates both of these constraints.

As it turns out, the convex hull of the two convex regions might not be representable in constraint matrix form, because some of the planes might not have the appropriate slope. We therefore use the convex shape formed by choosing the looser of corresponding constraints for each matrix.

The basic algorithm iterates through the constraints in the two regions a and b, constructing a new region c. The new region contains the looser of the two corresponding constraints. As we construct this new region, we create a list (called the *a-tight* list) of the constraints that were tighter in the first region than in the second. We also create a corresponding *b-tight* list. Constraints that are equal in the two regions are not added to either of these lists. If the *a-tight* list is empty at the end of the construction, then the second region is a subset of the first, so we return the first. Conversely, if the *b-tight* list is empty, then the first region is a subset, so we return the second. Otherwise we try to find a pair of constraints, one from the *a-tight* list and one from the *b-tight* list, such that the intersection of the c region and the logical negation of these two constraints is not void.

Since we promote all timing parameters to integers, and since all of our constraints are of the form $c_i - c_j \leq k$, the inverse of such a constraint can be given as a constraint of the form $c_j - c_i \leq -1 - k$. Thus, the obvious algorithm iterates through the *a-tight* list and the *b-tight* list and runs Floyd's algorithm to determine if the region is void. Since the length of each of the lists is potentially of order $O(n^2)$, and since Floyd's algorithm is $O(n^3)$, the worst-case complexity of this algorithm would be $O(n^7)$.

We improve on this. First, recall that, for every inequality in a canonical constraint matrix, there is some solution for the matrix with that inequality as an equation. This means that, if we have a single canonical matrix and a single additional constraint, we may determine in constant time whether the intersection of that region and that constraint is void. If the constraint is $c_i - c_j \leq k_1$, we locate the constraint of the form $c_j - c_i \leq k_2$ in the canonicalized matrix; if $k_1 + k_2 \geq 0$, then there is some point that satisfies both the matrix and the additional inequality.

Unfortunately, we have two inequalities, not one, and the above technique does not work with two inequalities. So we construct a canonicalized matrix from the original c matrix and one of the inequalities, and then inspect this matrix with the other inequality to determine if the result is void or not.

Adding a single inequality to a canonical constraint matrix results in a matrix that can be re-canonicalized in only $O(n^2)$ time. Thus, our resulting total algorithm has a worst-case complexity of $O(n^4)$, and it is usually much quicker than this. The algorithm is given in figure 62.

This algorithm can be further improved by combining the quick Floyd's algorithm with the test of the *b-tight* regions; the worst-case complexity remains the same.

Using aggressive coalescing improves performance for most examples. Occasionally it will have worse performance than the simple subset technique. In general, the larger the number of clocks, the greater the improvement the coalesce method provides.

The coalesce method does have one drawback that may not be obvious at first glance. Since it only considers regions two at a time, it might miss opportunities for combining regions. Consider a case with three clock variables. The first region encountered has $c_i \geq c_k, c_i \geq c_j$. Next, a region with $c_k \geq c_i, c_k \geq c_j$ is encountered; no merge between these two regions is possible. Finally, a region with, $c_j \geq c_i, c_j \geq c_k$ is encountered. This region cannot be merged with either of the previous two. Yet, the three regions can be merged into a single region.

```
matrix convex-union(matrix a, matrix b, int n) =
   matrix c = a
   list a_tight = (), b_tight = ()
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         if (a[i,j] < b[i,j]) then
            c[i,j] = b[i,j]
            a_tight = (i,j) :: a_tight
         else if (a[i,j] > b[i,j]) then
            b_tight = (i,j) :: b_tight
   if (a_tight == ()) then
      return b
   else if (b_tight == ()) then
      return a
   if (length(a_tight) > length(b_tight)) then
      swap(a_tight, b_tight)
      swap(a, b)
   for ((i,j) in a_tight)
      matrix d = c
      d[j,i] = -1-a[i,j]
      floyd-quick(d, i, j, n)
      for ((i,j) in b_tight)
         if (d[i,j] > b[i,j]) then
            return not_convex
   return c
```

Figure 62. An algorithm for calculating the convex union of two convex regions, if one exists.

Let us consider how the coalesce method works in practice with our execution example. After executing the trace $[a+, c+, f+]$, the constraint matrix is (with the timed tokens ordered $(2, 1, 5)$)

$$S_1 = \begin{pmatrix} 0 & 10 & 7 & 8 \\ -3 & 0 & -3 & -2 \\ 0 & 7 & 0 & 5 \\ 0 & 7 & 0 & 0 \end{pmatrix};$$

this matrix represents 110 discrete timed states. Later in the depth-first, after backtracking the last two firings and executing two more, the current path from root is $[a+, f+, c+]$, which yields a region of

$$S_2 = \begin{pmatrix} 0 & 10 & 7 & 7 \\ -3 & 0 & -3 & -3 \\ 0 & 7 & 0 & 0 \\ 0 & 7 & 4 & 0 \end{pmatrix},$$

representing 80 discrete timed states. Neither of these two constraint matrices is a subset of the other, but their union is convex, and it represents a total of 160 discrete timed states:

$$S_1 \cup S_2 = \begin{pmatrix} 0 & 10 & 7 & 8 \\ -3 & 0 & -3 & -2 \\ 0 & 7 & 0 & 5 \\ 0 & 7 & 4 & 0 \end{pmatrix}.$$

We replace the two previous regions in the state list with this single region, and use this single region to continue exploring. After firing $f+$ from this union region, we obtain the region

$$S_3 = \begin{pmatrix} 0 & 10 & 4 & 8 \\ -6 & 0 & -6 & -2 \\ 0 & 10 & 0 & 8 \\ 0 & 7 & 0 & 0 \end{pmatrix},$$

representing 85 discrete timed states. This new region is again no subset of any previous region, but this time it can be merged with $S_1 \cup S_2$, yielding

$$S_1 \cup S_2 \cup S_3 = \begin{pmatrix} 0 & 10 & 7 & 8 \\ -3 & 0 & -3 & -2 \\ 0 & 10 & 0 & 8 \\ 0 & 7 & 4 & 0 \end{pmatrix},$$

which represents all 196 discrete timed states seen so far. In addition, because the region $S_1 \cup S_2$ lies on the path back to the root, we need not consider any successors from that region because they will be a subset of the successors we find from the exploration of $S_1 \cup S_2 \cup S_3$. After depth-first traversal of the state space is completed, the total reachability graph has just seven timed regions—the minimum number capable of representing the reachable space.

When we use the coalesce method on our adverse example, the improvement is even more dramatic. With a k of ten and n of five, we see forty-two timed regions and 604 edges in only 6.7 seconds of CPU time; this is an improvement of about a factor of ninety over the basic subset method. The next size, with an n of six, finds 378 timed regions and 8,743 edges in only 667 seconds. With an n of seven, verification does not complete in eight hours of CPU time.

7.7 Partial Order Timing

None of the geometric timing techniques we have described so far takes advantage of the concurrency information available in a Petri net. These methods only calculate the timing for sequences of transition firings. For instance, in our operational example, after firing $a+$, the transitions $f+$ and $c+$ are concurrent, as are the transitions $f+$ and $b+$.

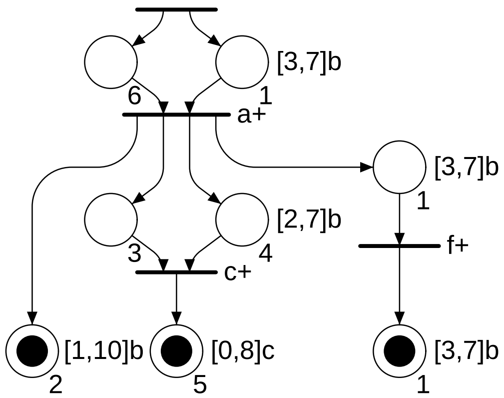


Figure 63. One process from the previous example.

As we saw in the section on coalescing regions, firing the sequence $[a+, f+, c+]$ leads to a different timed region than does firing the sequence $[a+, c+, f+]$.

Consider, however, the *process* constructed from the firing of the sequence $[a+, c+, f+]$, as shown in figure 63. A process is an acyclic, choice-free net created from a Petri net and a firing sequence. It is constructed by iterating through the firing sequence as followings: Initially, the process contains a single transition with output places corresponding to each token in the initial marking. We add the transitions in the same order as they occur in the firing sequence. For a transition in the firing sequence, we add a correspondingly labeled transition to the process. We also add the appropriate set of arcs to the process for that transition. The input arcs to the transition are connected from the most recently added places in the process corresponding to input places of the transition in the original Petri net. Finally, we add a new set of places corresponding to the output places of the transition in the original net and connect these places to the new transition.

Every place and every transition in the created process, except the first, corresponds to some place and some transition in the original net. Every place and every transition in the original net corresponds to zero or more places and transitions in the process. A process explicitly represents the concurrency in a particular firing sequence. That is, a particular process corresponds to many different firing sequences that differ only in the interleavings of concurrent transitions; every such firing sequence fires the same set of transitions and leads to the same final untimed state. Thus, the process shown in figure 63 corresponds to both the trace $[a+, c+, f+]$ and to the trace $[a+, f+, c+]$.

In general, if we have an acyclic, conflict-free Petri net with certain types of timing constraints among the nodes, we may calculate minimum and maximum separation between all pairs of the transitions in time proportional to $O(n^3)$ (using, not surprisingly, Floyd's algorithm again). If none of the places is timed, then the process is a partial order on the events. That is, an untimed place p introduces only the constraint *fire-time*$(\bullet p) \leq$ *fire-time*$(p\bullet)$. (Each place in a process has a single input and output transition.) Because the constraint places do not affect the enabledness or firing of transitions, such places introduce the same kind of causality relation. Behavioral places, on the other hand, affect the enabledness of transitions. For each behavioral place p in the resulting process with a timing requirement of $(min, max, behavior)$, two constraints are introduced. The first reflects the minimum separation, and the resulting constraint is *fire-time*$(p\bullet) -$ *fire-time*$(\bullet p) \leq -min$. The second reflects the maximum separation, and the resulting constraint is *fire-time*$(\bullet p) -$ *fire-time*$(p\bullet) \leq max$. All constraints introduced in this fashion for a given process must be satisfied. As we have shown earlier in the chapter, when we have a set of constraints that must all be true, Floyd's algorithm calculates the possible separations between events.

If we perform this operation on the above process, we find that $f+$ can follow $c+$ by at most five time units, and that $c+$ can follow $f+$ by at most four time units. In addition, $c+$ can follow $a+$ by between two and seven time units, and $f+$ can follow $a+$ by between three and seven time units.

From this information, we can easily calculate the full set of timed regions reachable with this process. Specifically, for two clocks c_i and c_j created by the firing of transitions t_i and t_j, the constraint on $c_j - c_i$, or entry a_{ij} in our constraint matrix, is the maximum separation from t_j to t_i. The ages of the clocks must be greater than or equal to zero, and the maximum age of the tokens in behavioral places is given by the place in which they reside. From this information, we can easily calculate a (non-canonical) constraint matrix of

$$\begin{pmatrix} 0 & 10 & 7 & \infty \\ 0 & 0 & -3 & -2 \\ 0 & 7 & 0 & 5 \\ 0 & 7 & 4 & 0 \end{pmatrix}.$$

Canonicalizing this matrix (which we can do in time n^2) results in

$$\begin{pmatrix} 0 & 10 & 7 & 8 \\ -3 & 0 & -3 & -2 \\ 0 & 7 & 0 & 5 \\ 0 & 7 & 4 & 0 \end{pmatrix},$$

the same matrix the coalesce method calculated for the union of the two traces represented by this process.

If the current process ends with n concurrent transitions, then with a single calculation we find the union of all $n!$ traces ending with permutations of those transitions.

Verification proceeds just as it does for the previous methods based on sequences, except that here, for each sequence, we calculate the corresponding process. With depth-first search, this is done incrementally. We also incrementally calculate a constraint matrix that stores the firing time relationship among the transitions. We then calculate the timed region corresponding to this process by adding the upper and lower bounds on the clocks; after normalizing this matrix, we have the full set of reachable states for that process. Compared to the previous algorithms, only steps three, five, six, and seven are changed (and step four is possibly skipped, depending on the result of step three).

3. If the current transition has no input constraint places, skip this and the next step. Otherwise, calculate a new current timing region, restricted to that subset of the previous timed region in which the transition we have selected was enabled. Canonicalize the resulting matrix.
5. Extend the current process with the new transition and output places.
6. Incrementally calculate a constraint matrix for the maximum separation between firing times of the transitions in the process.
7. Compute a timing region matrix by calculating the maximum difference between every pair of transitions in the process that is input to a timed place whose token has not been consumed. Initialize the lower bounds of the timing variables to zero and the upper bounds to the maximum allowed age (for behavioral places), or ∞ for constraint places.

We need to show that we will obtain the same set of reachable states using process-based region calculations as we do using sequence-based region calculations. Our process calculation can never introduce states that are not reachable by a sequence, since for any timed state in the process, some ordering of the transitions in the process is a sequence that generates the same timed state. In addition, because we consider the firing of every transition enabled in every reachable timed region, we will visit all successors and thus the entire timed state graph.

We now turn our attention to some efficiency considerations. The number of transitions in the process is equal to the length of the firing sequence plus one, and it increases with the depth of our search. Calculating the minimum separations between the occurrence times in our process, even with our incremental $O(n^2)$ approach, becomes prohibitively expensive as the firing sequence lengthens. In addition, we need one of these constraint matrices for each step; this would require a tremendous amount of storage during depth-first search.

To keep n bounded as the depth of our search increases, we need to determine what prefix, if any, of our process we can safely ignore. We can eliminate any transitions that we know will no longer affect our future calculations. In general, we can eliminate a variable from any set of equations or inequalities whenever we have the full set of equations or inequalities that use that variable. Since all constraints introduced through the firing of a transition are associated with places connecting the new transition to the old, once a transition in our process no longer has any output places, it is eliminated from our constraint matrix. Thus, our n is—at most—the number of tokens in the net at any given time, plus one for the current transition.

In the section on specializations of Floyd's algorithm, we will introduce some more optimizations that make partial order timing even more efficient.

We may still safely use the subset method of controlling our search. When we do this with our execution example, the final reachability graph has eight timed regions and eighteen edges. Alternatively, we may generate an exact reachability graph with respect to processes by requiring an exact region match during our search; for our example, this results in eleven regions and twenty-two edges.

This technique has a more expensive transition firing computation, so for the simplest examples with no or little concurrency, it will be slower; but, it dramatically reduces the number of interleavings considered when there are concurrent transitions. It works much faster on our adverse example than the other techniques; indeed, for this example, the run time is polynomial in n and constant in k. For example, verifying with an n of six and k of ten requires about a tenth of a second of CPU time, finding at the end only a single timed region and some forty-eight edges, an improvement of more than six thousand over any previous technique. It also works for much larger n and k; setting n to ten and k to one hundred yields a single timed region and 120 edges in less than one half second of CPU time. Setting n to forty and k to ten thousand again yields a single timed region and 1,680 edges in only 55.2 seconds.

For the Seitz queue element, partial order timing yields a reachability graph with only 410 regions and 710 edges in 2.0 seconds of CPU time. There are only 344 untimed states, so each untimed state has an average of 1.2 timed regions, which is close to optimal.

7.8 Pruning the Interleavings

The last improvement we present is a method for eliminating redundant interleavings from our partial order method. In figure 65, there are two possible interleavings of sequences that lead to this identical process. Exploring both of them is redundant.

We define a *canonical firing sequence* for every process: Number the transitions in the original net in some arbitrary way so that each transition gets a unique number. As a process is constructed by copying net transitions, copy the unique number of the net transition over to the process transition. Define the set *front* of frontier transitions in the

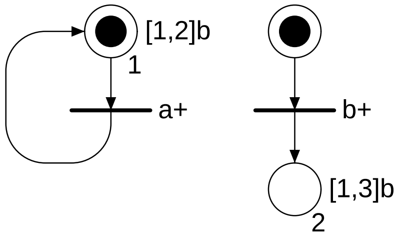


Figure 64. A subtle problem with pruning interleavings.

process as those transitions that have no successor transitions; that is, none of the output places of these transitions have any output transitions. There must be at least one such transition if the process has any transitions at all. Select the transition with the largest unique identifier; that will be the last transition of the canonical sequence. Remove this transition and its output places from the process, yielding a new, smaller process, and add the canonical sequence for this process to the front of the transition we just removed. This is the canonical firing sequence for the process.

The first technique to prune redundant interleavings that we might consider is to fire just those sequences that are equivalent to the canonical sequence of the resulting process. This fails in two ways, one obvious, one subtle. Obviously, the canonical sequence might not be a firing sequence of the net, because timing constraints might cause a particular ordering to be impossible. We might work around this with a slightly more complex definition of the canonical sequence. The subtle problem is more troublesome; we might actually miss some states because of premature backtracking. Consider the net shown in figure 64. Transitions $a+$ and $b+$ are concurrent in this net. We shall define a canonical sequence as $[a+, b+]$; this sequence is allowed by the net.

The initial region is described by

$$\begin{pmatrix} 0 & 2 \\ 0 & 0 \end{pmatrix},$$

indicating that the timed token in place one can have an age from zero to two. We begin depth-first search by firing $a+$; the resulting timed region is the same, so we backtrack. Note that we did not consider the sequence $[a+, b+]$ because of this state-based backtracking.

Next, we fire $b+$. The region reachable by this firing has a matrix of

$$\begin{pmatrix} 0 & 2 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix}$$

(with the places in the order $(1, 2)$), indicating that both tokens can have an age between zero and two, but that the token in place two must be older than the token in place one.

From this state, we do not consider the firing of $a+$, since $[b+, a+]$ is not the canonical sequence of this process. So we backtrack.

If we do execute $a+$ from this state, or if we execute the sequence $[a+, b+]$, we would find the region

$$\begin{pmatrix} 0 & 2 & 3 \\ 0 & 0 & 2 \\ 0 & 2 & 0 \end{pmatrix},$$

which is larger than the previous region. Thus, pruning interleavings according to some canonical firing sequence, even if it satisfies the timing requirements of the net, can cause us to miss some processes and thus some states.

We solve this problem by storing a representation of each process we have seen. As we are about to fire a transition, we check whether the resulting process has already been explored. If so, we backtrack. One concise representation of the process is the canonical firing sequence. While such a sequence might be long, most of the time it is just a single transition or a few transitions tacked on to the end of the previous canonical sequence, so the two canonical firing sequences can share most of the same storage. For the same reason, hashing and comparison operations are quick. Because we only reject a process if we are sure we have seen that precise process before (and not just the resulting timed states), we are sure to explore the entire state space.

If we do not prune the interleavings, we would compute precisely the same state space, although we might visit a particular process many times. Instead of performing all of the timing calculations for the later visits, therefore, we simply examine our hash table of canonical sequences to determine if that process has already been visited. For a simple net with just n concurrent transitions, each of which can fire only once, the partial order method would require us to perform our timed successor calculation $n2^{n-1}$ times. Using partial order pruning, we check our canonical sequence hash table $n2^{n-1}$ times, but only perform the timed successor calculation $2^n - 1$ times.

Interestingly, the adverse example does not profit at all from this form of partial order pruning, since the subset method already eliminates almost all backtracking during the depth-first search. For many of our circuit examples, however, pruning the interleavings does improve performance.

7.9 Specializations of Floyd's Algorithm

In this section we improve the efficiency of recanonicalization of the matrices. This operation is the core operation in our timing verification. These improvements are simply incremental variations of Floyd's algorithm that calculate a canonical matrix from a slightly modified canonical matrix. Our knowledge of the properties of the modifications allows us to determine that most of the comparisons in Floyd's algorithm will always evaluate to false, so that the comparison need not be made.

All of the specializations are based on the observation that adding a new timed variable and a set of constraints relative to that timed variable to an already canonical set of inequalities can be done in time proportional to n^2. We will describe some of the specializations we require.

Consider step three in our timed region successor calculation. It is performed before firing a transition; it eliminates from the current region those timed states in which the transition is not enabled. This is done by introducing the single constraint indicating that the age of the behavioral token to be consumed (if any) must be greater than or equal to the minimum age allowed by that place. We introduce this constraint by decreasing the $a[b,0]$ entry, where b is the row and column number associated with that place. Since the numbering is arbitrary, we can consider the nodes in any order:

```
sequence p = {0, 1, ..., n-1}
for (k in p)
   for (i in p)
      for (j in p)
         if (a[i,j] > a[i,k] + a[k,j]) then
            a[i,j] = a[i,k] + a[k,j]
```

Floyd's algorithm is idempotent when the region is not void; when running Floyd's algorithm on a canonical matrix, the inner inequality will never be satisfied. We reorder Floyd's algorithm by using the sequence

```
sequence p = {1, 2, ..., b-1, b+1, ..., n-1, b, 0}
```

Now consider the comparison in Floyd's algorithm. The only change we have made to the matrix is that element $a[b,0]$ has been decreased. But we do not even read that element until the last two loops of k, so we know that the first $n-2$ loops will not change the matrix. Thus, we reduce the outer loop to

```
for (k in {b,0})
   for (i in p)
      for (j in p)
         if (a[i,j] > a[i,k] + a[k,j]) then
            a[i,j] = a[i,k] + a[k,j]
```

Even here, for $k = b$, the inequality can only be true if $j = 0$, and thus the only elements of a that can change in the first outer loop are in the first column. The algorithm now becomes

```
for (i in p)
   if (a[i,0] > a[i,b] + a[b,0]) then
      a[i,0] = a[i,b] + a[b,0]
for (i in p)
   for (j in p)
      if (a[i,j] > a[i,0] + a[0,j]) then
         a[i,j] = a[i,0] + a[0,j]
```

Furthermore, the only way the second inequality might become true is if the first element of the row is decreased by the first loop, or if $i = b$. Thus, we improve things further:

```
for (j in p)
   if (a[b,j] > a[b,0] + a[0,j]) then
      a[b,j] = a[b,0] + a[0,j]
for (i in p)
   if (a[i,0] > a[i,b] + a[b,0]) then
      a[i,0] = a[i,b] + a[b,0]
      for (j in p)
         if (a[i,j] > a[i,0] + a[0,j]) then
            a[i,j] = a[i,0] + a[0,j]
```

This specialization of Floyd's algorithm is $O(n^2)$.

In step seven, we advance time by increasing the maximum constraints on the clock variables. Since the only entries that are changed are the maximum values of clocks, and each of these maximums is either increased or unchanged, this canonicalization requires a different specialization of Floyd's algorithm.

The first observation to make when deriving this specialization is that in any canonical matrix, if we increase a set of values and then recanonicalize, the values that were unchanged are the same in the new matrix because they were already at their minimum values, and increasing other matrix elements does allow them to be reduced further. Therefore, the only possible change is to row zero of the matrix. With this observation, Floyd's algorithm reduces to

```
for (k=0; k<n; k++)
   for (j=0; j<n; j++)
      if (a[0,j] > a[0,k] + a[k,j]) then
         a[0,j] = a[0,k] + a[k,j]
```

This is obviously $O(n^2)$.

Other specializations can be derived in a similar manner. Our partial order method deserves some careful attention because we must calculate a constraint matrix over a process, which can be much larger than the number of timed tokens in a net. We have already seen how we can prevent the number of transitions of that process from growing without bound. Now we refine the algorithm to save both time and space.

The most expensive computation in partial order timing is calculating the resulting matrix for a process extended with a single additional transition from the matrix for the shorter process. A specialization of Floyd's algorithm forms the starting point for our improvements. For simplicity, we number the transitions in the process according to their order in the firing sequence; this way, the last transition added is transition number $n-1$. We extend the previous matrix with an additional row and column, and initialize all values to ∞ except for the diagonal, which we initialize to zero.

Finally, we run Floyd's algorithm. The first $n-1$ iterations of the k loop only affect the new row and column we introduced, because the previous matrix was already canonical; if $i < n-1$ and $j < n-1$, the inequality must be false. We therefore rewrite Floyd's algorithm as follows:

```
for (k=0; k<n-1; k++)
   for (i=0; i<n; i++)
      if (a[i,n-1] > a[i,k] + a[k,n-1]) then
         a[i,n-1] = a[i,k] + a[k,n-1]
   for (j=0; j<n; j++)
      if (a[n-1,j] > a[n-1,k] + a[k,j]) then
         a[n-1,j] = a[n-1,k] + a[k,j]
for (i=0; i<n-1; i++)
   for (j=0; j<n-1; j++)
      if (a[i,j] > a[i,n-1] + a[n-1,j]) then
         a[i,j] = a[i,n-1] + a[n-1,j]
```

We improve it further by noting that the only constraints introduced in row and column $n-1$ come from transitions connected by input places. Thus, we use

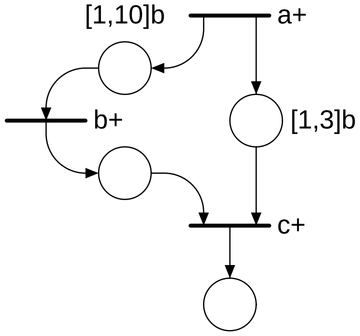


Figure 65. A process illustrating how previous constraints can be tightened.

```
sequence p = map predecessor (input-places-of transition n-1)
for (k in p)
   for (i=0; i<n; i++)
      if (a[i,n-1] > a[i,k] + a[k,n-1]) then
         a[i,n-1] = a[i,k] + a[k,n-1]
   for (j=0; j<n; j++)
      if (a[n-1,j] > a[n-1,k] + a[k,j]) then
         a[n-1,j] = a[n-1,k] + a[k,j]
for (i=0; i<n-1; i++)
   for (j=0; j<n-1; j++)
      if (a[i,j] > a[i,n-1] + a[n-1,j]) then
         a[i,j] = a[i,n-1] + a[n-1,j]
```

This routine is obviously of complexity $O(n^2)$, but we can improve it further. The first two loops above calculate the minimum and maximum separations between the new transition and all of the old transitions. The third loop tightens any old constraints that need tightening because of the appearance of this new transition.

This third loop bears further study. It is not obvious how the firing of a new transition can further constrain the separation between transitions that have already occurred. If the process were an orbital net with the firing semantics we have defined, such tightening would not be possible.

But the process is not a net with our semantics. It is just the unfolding of a particular execution of a firing sequence. Consider the process illustrated in figure 65. Without the execution of $c+$, the transition $b+$ can follow $a+$ by any amount of time between one and ten units, inclusive. When $c+$ occurs, we also know that $c+$ cannot follow $a+$ by more than three units. Yet, because $c+$ must occur after $b+$, this means that $b+$ also cannot occur more than three time units after $a+$. Thus, if this process occurs, we know that we are only exploring the processes of the net for which $b+$ follows $a+$ by less than three units of time; presumably, there are other processes for other firing sequences that take care of the other cases.

We have shown why that last loop, the one that dominates the algorithm, is necessary. In the experiments we have performed, however, the last loop usually does not affect the matrix; it only occasionally affects the matrix. A quick test determines when it will have no effect. Let us imagine that we are performing the last loop. We are currently considering some i such that $i \notin p$, and any j. The inner condition examines $a[i, n-1]$. Because $i \notin p$, this element is initialized as ∞, and the first loop of the algorithm sets it to $a[i, i'] + a[i', n-1]$ for some $i \in p$. Thus, if the inequality in the last loop is true for $i \notin p$, we must have

$$a[i, j] > a[i, i'] + a[i', n-1] + a[n-1, j].$$

Since i and j are less than $n-1$, and the original matrix was canonical and its entries are as of yet unchanged, we also have

$$a[i, i'] + a[i', j] \geq a[i, j].$$

When we add these two inequalities, we get

$$a[i', j] \geq a[i', n-1] + a[n-1, j]$$

for some k in p. We now perform the same operation over the columns as we did over the rows, to change a j that might not be in p to some j' that is definitely in p:

$$a[i', j'] \geq a[i', n-1] + a[n-1, j'].$$

Thus, if any of the entries of the original matrix will be tightened, then some entry will be tightened where both i' and j' are in p. This gives us a quicker test to see if we must perform the last loop:

```
for (i in p)
   for (j in p)
      if (a[i,j] > a[i,n-1] + a[n-1,j]) then
         perform-full-third-loop
         break
```

Since p has a length equal to the number of input places to our transition, and since normally the last loop is unnecessary, this yields a substantial improvement.

Because the third loop is rarely required during execution of our net, we save space as well. When the third loop is not needed, the new matrix is just the old matrix extended with a new row and column (and with a few rows and columns perhaps deleted). We calculate and store just the single new row and column for most transitions in our process, yielding a substantial savings in space.

Chapter 8

Data Flow

None of us really understands what's going on with all these numbers. —David Allen Stockman, 1946–

Partitioning a circuit design into separate control and computation portions is a powerful and necessary technique, for both manual and computer-aided design. The techniques and tools we have presented so far are primarily concerned with control. In this and the next chapter, we discuss the integration of data flow and computation into Orbits.

It is possible to handle data flow as we have dealt with control. The electrical components used to implement the data path of a circuit are the same components, with the same type of behavior, as those used to implement control. A flip-flop behaves the same whether it is used as part of a controller or part of a data path.

For naturalness and efficiency we distinguish the data flow and deal with it in a different manner. A thirty-two bit adder composed of thirty-two full adders can be described with individual detail to each individual wire of the input and output data paths and carry chain. Perhaps we can even verify such a system—although the state space is huge. But this adder is a well-defined primitive component of a larger data-path; it is more natural both for the designer and for the computer-aided design tool to consider it as a computational unit that takes precisely two inputs and generates a single result. This is a different situation than with control, where typically every bit of state is important and cannot be easily combined with other state bits.

A natural behavioral semantics for data flow differs from a natural semantics for control. With control, every individual positive or negative edge of every state bit is of concern, since every such edge is essentially a state transition. Yet, for data flow, we are primarily concerned with when a particular value on a set of wires is valid and when it is not; each individual value change of the individual wires comprising a data bus is not of import. Thus, the valid-to-invalid and invalid-to-valid transitions are the important ones.

In addition, it is much more efficient to consider data flow in its own right. The state space of a single thirty-two bit value is in excess of four billion states—exploring this state space is beyond the ability of most current finite state verification tools and techniques. When this state space is exploded by the size of a typical data path—with many independent registers—it is clear that standard finite-state techniques are inadequate for all but the smallest designs, despite some recent successes with binary decision diagrams [21].

When timing is considered, the inefficiencies become even more extreme. When each bit of a data path is treated as its own variable delay, each active data path element introduces a large number of additional clock elements, and the timed state space explodes tremendously. Yet, the value represented by a set of data path bits cannot be considered valid until each bit is valid. While with control it is important to consider every possible delay from the minimum to the maximum for each control element, for data flow, it is a reasonable approximation to use the minimum delays for valid to invalid transitions and the maximum delays for invalid to valid transitions.

In this chapter, we will discuss how we represent, specify, and simulate simple data flow aspects in Orbits. We will extend our syntax and operational semantics for data flow, and give an example of verification with data flow. We will describe how communication of data flow values between concurrent agents can be performed over *keys*. In the next chapter, we will consider data flow values on wires, and show how data flow elements can be modeled and simulated.

While Orbits includes the ability to specify, represent, model, and simulate data flow, we have not yet extended our behavioral semantics in such a way that full formal verification can be performed. This is an important extension to Orbits left for future research.

8.1 Net and Language Syntax

In chapter three we introduced values, operations, conditionals, and bindings of identifiers to values for the purposes of state machine description and initial value propagation through partial evaluation. In these contexts, we required that each such expression be a simple expression, that is, known at the time of evaluation for orbital net construction. To use these operations for specifying data flow, we lift this restriction and define a translation of data flow into orbital nets.

For values and operations, we adopt a standard directed acyclic graph (DAG) description, also known as a DFG (data flow graph). We extend our orbital net formalism with the four additional sets F, DE, R, J, and one new function *func*. The sets F, R, and J are

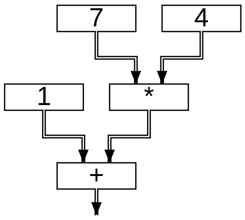


Figure 66. The orbital net representing the expression `(+ 1 (* 7 4))`.

nodes that represent *functional units*, *registers*, and *register joins*, respectively; the set DE represents *data flow edges* that connect them, and the function *func* returns the specific function that a particular functional unit computes.

A data flow edge is a four-tuple (*innode*, *inpos*, *outnode*, *outpos*) representing the input and output nodes and the position of the connection in the sorted list of arguments. For all connections except inputs and outputs from function nodes, the positions are zero. For edges connecting to function nodes, the position indicates which input or output element to that function the data flow edge represents. The pair of nodes in each data flow edge is a member of the set $R \times J \cup (F \cup J \cup P) \times (F \cup R \cup T \cup P)$; that is, data flow edges can connect registers to joins, or have their source connected to a functional unit, a join, or a place and their sink connected to a functional unit, a register, a transition, or a place. We will discuss connecting data flow edges and places in the next chapter. Data flow edges are drawn in our figures as double lines.

A functional unit from F is a node that has zero or more data flow edges as inputs and has one or more data flow edges as outputs. The edges carry data flow values. The values that flow through these data flow edges are the integers plus the special value `unknown`. Constants are represented by functions with no inputs. Applying *func* to a node from F returns either an integer constant, the special value `unknown`, or one of the standard arithmetic operations `+`, `-`, `*`, `/`, `<<`, `>>`, `=`, `>`, `<`, `>=`, `<=`, `<>`, `odd`, `even`, `not`, `and`, or `or`. This list can be easily extended, and we will often assume that it is extended for obvious functions such as `1+`. The trivial expression `(+ 1 (* 7 4))` would yield the orbital net given in figure 66, except that the partial evaluation performed on this expression would reduce the result to twenty-nine. Functional units are drawn with rectangular boxes with a label indicating the value returned by *func* for that box.

A register from R represents an assignment of a value to a storage node. A register has a single data flow input and a single data flow output; when the register fires, the value at its input is passed to the storage node on its output. These storage nodes are the register joins from J; the output of each register node must be a single join node, and each join node must have one or more register nodes as input. The join nodes are the only nodes (except for wires, discussed in the next chapter) that hold values. In particular, the only type of recursion we allow is tail recursion, since it can be implemented by assignment and control flow; we do not include the implicit stack that more general recursion would require. We draw registers with a box with a small triangle to the left (as though it were an edge-triggered flip-flop); we draw join nodes as large black dots.

The assignment represented by a register node implies some time at which that assignment takes place; that is, the assignment itself is an event or a change of state. For this reason, the firing of a register is controlled by a transition; a Petri net edge from a transition to a register indicates that when that transition fires, the register also fires at the same time. Thus, we extend the PE set of the orbital net to also include edges from the set $T \times R$.

We described labeled `let` in chapter three. Specifically, we described the version with no parameters. We can easily extend it to handle bindings as an unlabeled `let` does.

`(let` *label* `((`*id0 exp0*`) (`*id1 exp1*`) ... (`*idn expn*`)) .` *body*`)` Evaluates all of *exp0* ... *expn* in parallel and associates a register and a join node with each expression and identifier to represent the initialization and the stored value, respectively. A transition is allocated and connected to all of these registers; this represents the initialization of the values. During evaluation of *body*, it associates the identifier *label* with output place of this new transition. Every invocation of *label* within body is treated in exactly the same way as the initial values were, with new registers and a new transition to transfer the new values into the join nodes originally created, and with the output place of the new transition merged with the place associated with *label*.

A labeled `let` allows us to create loops, as in the following expression which describes a data flow loop that counts from zero on up:

```
(let deficit ((i 0))
   (deficit (+ i 1)))
```

The orbital net resulting from this expression is given in figure 67. Observe how the separation between control and data flow shows up in orbital nets much like it does in circuits.

Loops are often more useful if they can be exited; in general, we need a method for the data flow to affect the control. For this, we allow a transition to also have one or more data flow edges as inputs. If the data flow edges all have a non-zero value, then the transition is enabled; if any of the data flow edges have a value of zero, then that transition is not enabled. The primitive functions that perform comparisons or logical connectives use zero for false and one for true. They also have two outputs: the first one is the positive output

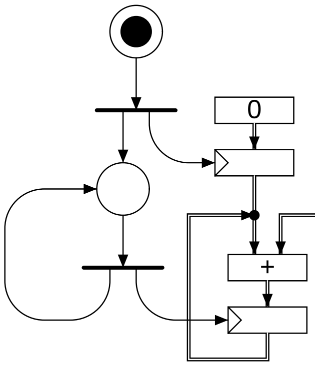


Figure 67. A simple loop using orbital nets.

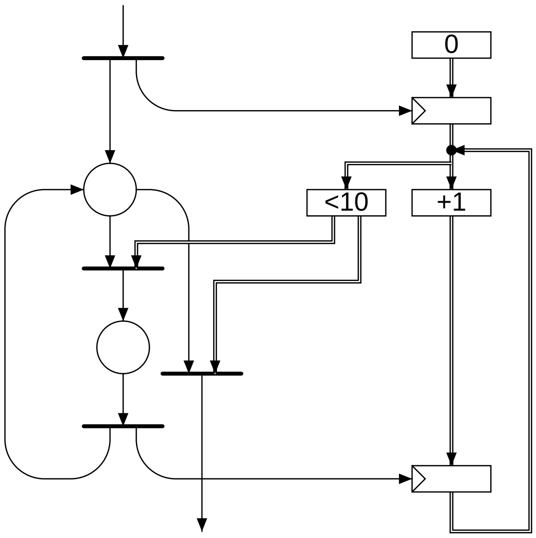


Figure 68. A loop that counts from zero to nine.

the reflects the result; the second is the negative output that is the logical negation of the first output. Having a positive and a negative output for Boolean conditions keeps our graphs cleaner and simpler. With this control, we easily describe finite loops:

```
(let for ((i 0))
   (when (< i 10)
      (for (+ i 1))))
```

The resulting net is shown in figure 68.

As another example, we describe an algorithm such as Euclid's algorithm for greatest common divisor, as shown in figure 69. Redundant registers (those whose input is from the same join node that their output is connected to) are automatically eliminated by Orbits.

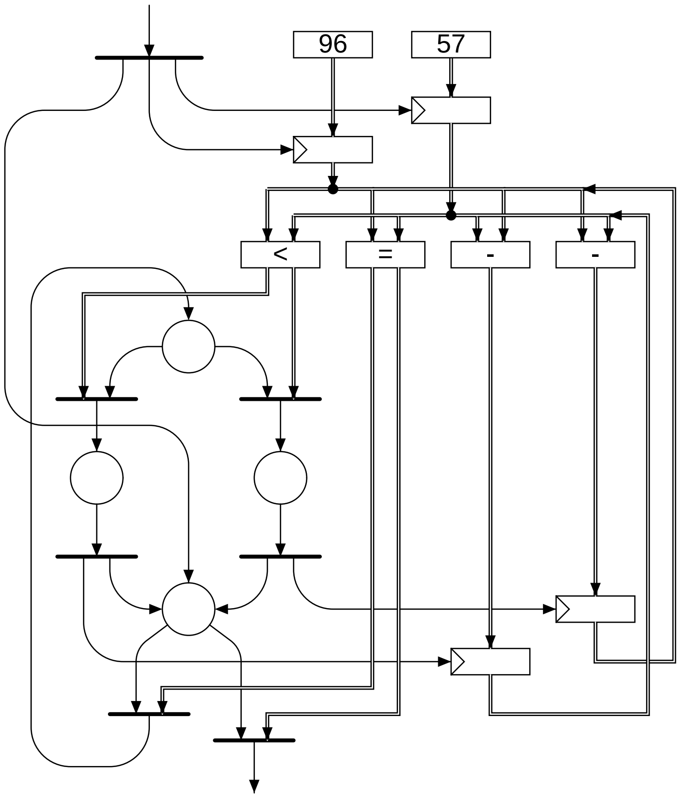


Figure 69. Representation of Euclid's greatest common divisor algorithm.

```
(let mgcd ((a 96) (b 57))
   (if (= a b) a
      (if (< a b)
         (mgcd a (- b a))
         (mgcd (- a b) b))))
```

It is often difficult to describe the operation of a circuit in such a functional style, since most circuits tend to have explicit registers and side effects. For this reason, the following primitive functions allow the declaration of storage and the use of assignment statements.

`(make-reg)` Declares data flow storage by allocating and returning a new join node. This is typically bound to an identifier using a `let` statement. The result of `make-reg` from the perspective of net construction is a data flow edge from the allocated join node representing the last value stored into that join node by the `set!` primitive.

`(set!` *var expr*`)` Allocates a new transition and a register enabled by that transition that gates the value returned by *expr* into the join node represented by *var*.

The *var* must be the value returned by the `make-reg` primitive. This models an assignment statement.

With these two primitives, it is easy to describe common data flow algorithms. For instance, we write Dekker's algorithm for mutual exclusion [45] as follows:

```
(def (dek-half () (req ack) myflag otherflag turn otherguy)
   (process
      (set! myflag 1)
      req+                  ; request
      (let loop ()
         (when (= otherflag 1)
            (when (= turn otherguy)
               (set! myflag 0)
               (let loop ()
                  (when (= turn otherguy)
                     (loop)))
               (set! myflag 1))
            (loop)))
      ack+                  ; enter crit
      req-                  ; leave crit
      (set! turn otherguy)
      (set! myflag 0)
      ack-))
```

Here, `myflag`, `otherflag`, and `turn` are shared global variables. The wires `req` and `ack` indicate, with standard four-phase signaling, request for and entrance into the critical section. The full system has two clients:

```
(def (dekkers () (a1 a2 r1 r2))
   (let ((turn (make-reg))
         (f0 (make-reg))
         (f1 (make-reg)))
     (set! turn 1)
     (set! f0 0)
     (set! f1 0)
     (par
        (dek-half r1 a1 f0 f1 turn 1)
        (dek-half r2 a2 f1 f0 turn 0))))
```

In order to test that mutual exclusion is satisfied, we write a simple specification that describes successful behavior.

```
(def (mutex-spec () (a1 a2 r1 r2))
   (process (choice (a1+ r1-) (a2+ r2-))))
(def mspec (mirror mutex-spec))
(def spec (receptive mspec))
```

Verification is then

```
(verify
   (with-wires global (r1 r2 a1 a2)
      (par
         (dekkers a1 a2 r1 r2)
         (spec a1 a2 r1 r2))))
```

and we find that, indeed, Dekker's algorithm satisfies the mutual exclusion requirement.

Because Dekker's algorithm only reads a single global variable for each control flow decision, it is not necessary to distinguish the reading of the global variable from the selection of the appropriate control flow branch; we consider the branch decision to have been made precisely when the variable is read. For algorithms that read two or more global variables at separate times when making a control decision, there is no longer a single access to associate the decision point with. For instance, if `a` and `b` are shared global variables, the expression `(and a b)` will only be true if both variables are true at the same time. If an implementation of an algorithm reads the global variables at different times, then it is possible for the algorithm to see them as both true, even if both were not true at precisely the same time. Thus, it is often useful to specify the reading of a variable as an event that can be ordered (or not ordered) with respect to other events. The `atomic-read` primitive provides this facility.

`(atomic-read` *expr*`)` Allocates a new transition and a new register triggered by that transition that stores the value of *expr* at the time the transition fires. This reflects the reading of a shared variable at a particular instant of time.

For instance, the expression `(and a b)` can only be true if both `a` and `b` are true at the same point in time. The resulting net is in figure 70. The expression `(and (atomic-read a) (atomic-read b))`, on the other hand, models the reading of the two variables in some arbitrary order, and it can be true if there is some point at which `a` is true and some possibly distinct time at which `b` is true; the corresponding net is shown in figure 71. The expression `(let ((v (atomic-read a))) (and v b))` models the reading of `a` and `b` in that order, and it can only be true if `a` is true at some point and `b` is true at some later point; this net is shown in figure 72.

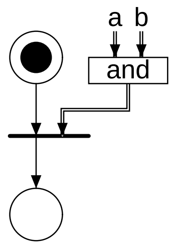


Figure 70. The net resulting from translation of `(if (and a b) ...)`.

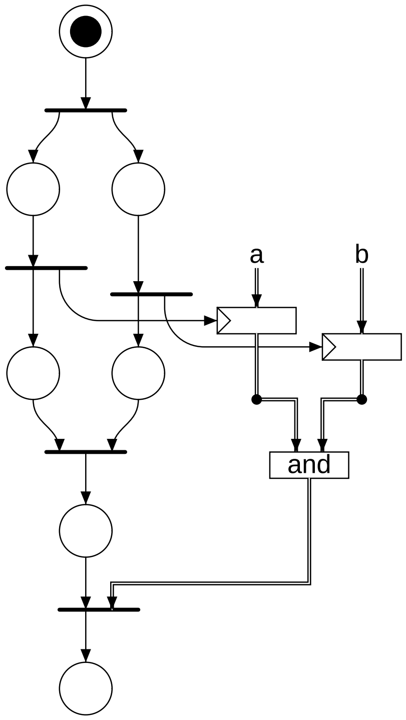


Figure 71. The net resulting from translation of `(if (and (atomic-read a) (atomic-read b)) ...)`.

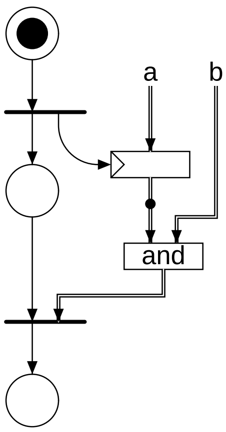


Figure 72. The net resulting from translation of `(let ((va (atomic-read a))) (if (and va b) ...))`.

8.2 Keys

One of the premises of Orbits is describing complex systems by decomposing them into separate descriptions of different aspects of their behavior. For instance, as the arbiter with reject example showed, it is possible to describe the interfaces separately from other requirements on the behavior. When we introduce data flow to specifications, we would like to retain this ability. Specifically, if a particular interface is used to obtain data values from or send data values to the environment, we would like to be able to describe this separately from the part of the circuit that operates on those data values.

While it is possible to use the primitives we have described so far to perform such a separation, it is awkward because of the way access to the shared variable needs to be controlled. For this reason, this section introduces keys, which are abstract communication channels over which data values are communicated among concurrent agents. The operations on the key construct the minimal appropriate orbital net that represents the communication of data values and all necessary synchronization.

Keys are similar to the labels used by CSP as operands to `!` and `?`, except that keys are dynamically allocated (during net construction) and can be passed as parameters. The primitive `make-key` is used to allocate a new key.

`(make-key 1)` Allocates and returns a new key object. This key object is normally bound to an identifier by a `let` statement and passed as a parameter to two or more separate agents that use it for communication. The parameter `1` is supplied to indicate the number of values transferred in each operation; currently only the value `1` is supported.

Keys constitute the second type of action objects in Orbits (the first type is wires). Like wires, keys are objects on which synchronization of concurrent agents takes place. Like wires, such synchronization is controlled by the group hierarchy. Unlike wires, operations on keys always occur in a matched pair, with a single send operation and at least one, but possibly more than one, receive operation. The operations that are performed on keys are `!`, to send a value, and `?`, to receive a value.

`(!` *key expr*`)` Allocates a new transition that models the sending of the data flow value resulting from the evaluation of *expr* down the channel represented by the key *key*. This transition is synchronized with one or more transitions created by `?` operations on the same key.

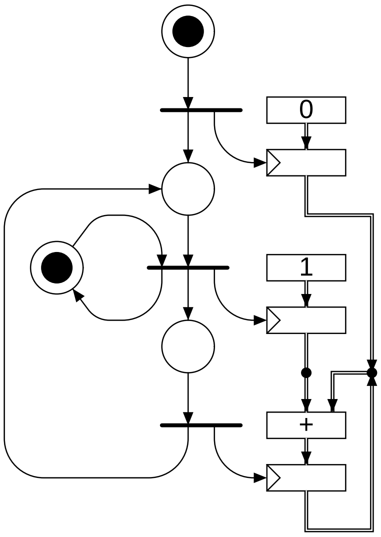


Figure 73. Two agents communicating over a key.

`(? key)` Allocates a new transition that models the reception of a value from the channel *key* and returns the received data flow value. This transition is synchronized with exactly one transition created by a `!` operation on the same key.

The communication of the value is modeled with a single register storing the communicated value onto a single join node. As a simple example, the following description is of two agents. The first keeps sending the value one to the second. The second accumulates the total of the received values.

```
(let ((k (make-key 1)))
   (par
      (process (! k 1))
      (complete ()
         (let loop ((s 0))
            (loop (+ s (? k)))))))
```

The resulting graph is shown in figure 73. The key communication is implemented by the middle transition and the middle register; no additional net node types are required.

For wires, we use epsilon places to simplify synchronization, especially when there are a large number of occurrences of a particular event in a group. Epsilon places can not be used for key synchronization because we have not yet found a construction that would result in the appropriate data flow semantics. So, for all such synchronization, the transition replication and merging construction is used instead.

To structurally enforce the nature of data communication using keys, we require all `!` operations on a specific key to occur within the same group. This way no two `!` operations can ever be synchronized with each other; such a construction does not have a clear meaning or obvious use.

In the next chapter, when we introduce data flow on wires, we will examine another example illustrating the descriptive power of keys for abstracting data flow communication between agents.

8.3 Operational Semantics

We have given informal descriptions and examples of how data flow is integrated into Orbits. In this section, we give a more formal description and discuss some subtleties that arise when defining an appropriate semantics.

If we store data flow values on join nodes, then obviously the value stored must be part of our state. We therefore extend markings to include the value stored at each join node. We also define the initial marking to include the value `unknown` for each join node.

We must also specify the value of all other data flow edges. For edges that come out of join nodes, we use the value stored on that node. For edges that come out of register nodes, we (arbitrarily) define the value to be the same as that stored on the join node that the data flow edge is connected to. All other data flow edges must originate in function nodes; we define the output value to be the result when the function is applied to the values on the inputs to those function nodes. The functions are extended to handle `unknown` inputs; if any input us `unknown`, so is the output. We prohibit cycles in function nodes, and there is no way to construct such a cycle with the textual language we define. Thus, the value on all data flow edges can be calculated from just the values stored in the join nodes.

A transition with no input data flow edges is enabled as before. If a transition has one or more input data flow edges, the value on all such edges must be not zero and not `unknown` in order for the transition to be enabled. If a transition is otherwise enabled except that one or more data flow edges have the value `unknown`, we flag this as an error, although other definitions are possible.

The firing of transitions and steps occurs precisely as before, except for those transitions that enable registers. First, all operations that were defined before for firing occur. Then, all values at the inputs to all registers that are enabled in the step are temporarily associated with that register (but not stored on the data flow edges). Next, the value in each join node that is an output of an enabled register is replaced by the value that was associated with that register, and the association is forgotten. Finally, new data flow values for all data flow edges are calculated based on the new values in the join nodes.

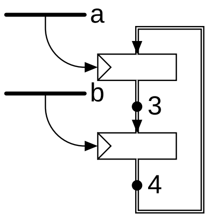


Figure 74. A net illustrating a problem with concurrently enabled transitions in a data flow context.

This firing semantics has a slight problem. Consider the situation where two registers whose output is the same join are concurrently enabled and fired in a single step; what value is then given to the join node? Recall that we require the firing of a step from a marking to lead to a single well-defined marking; for this reason, we cannot arbitrarily choose between the two values. Another alternative is to define the resulting value as `unknown`. This violates the interleaving property, the requirement that the result of firing of two concurrent transitions in a single step is the same as the result of firing the two transitions in either order.

Indeed, a similar situation can arise even when each join node has only a single input register. Consider the net given in figure 74. Transitions a and b are concurrently enabled. Firing both simultaneously swaps the values on the join nodes. Firing a and then b leaves both join nodes with the value four; firing b and then a leaves both join nodes with the value three.

The solution to this problem is to disallow it; no two transitions are concurrently enabled if they either read or modify, directly or through some number of function nodes, the same join node. This is similar to the situation with tokens in places, in that no two transitions are concurrent if they share an input place.

More precisely, we state that a data flow edge depends on a join node if it is an output edge of that join node. A data flow edge that is the output of a register depends on the join node to which it connects. A data flow edge that is the output of a function node depends on the union of the join nodes that the input data flow edges to that function node depend on. A transition reads a join node if and only if any of the data flow edges that enable that transition depend on that join node, or if the input data flow edge to a register fired by that transition depends on that join node. A transition writes a join node if the output data flow edge of a register fired by that transition depends on that join node. Thus, the set of join nodes read or written by a particular transition can be statically determined. Any two transitions that read or write the same join node must be considered to be in conflict.

With this modification to the operational semantics, the interleaving property is satisfied.

8.4 Behavioral Semantics

We retain our behavioral semantics as previously defined based on the visible event occurrences on wires. As our examination of Dekker's algorithm shows, this semantics is useful even without the explicit inclusion of data flow. Because it is almost always easy to construct a failing firing for a given data flow occurrence (such as a mismatch of values at particular stages), this is generally useful. On the other hand, the surfeit of internal transitions introduced by control flow aspects of a specification make mirroring and conformation somewhat more difficult. We do not extend our behavioral semantics to consider the data flow aspects in a more detailed manner, leaving this as a potentially fruitful area for future research.

Chapter 9

Data Flow on Wires

I lisp'd in numbers, for the numbers came. —Alexander Pope, 1688–1744

This chapter discusses how Orbitsmodels wires that carry data. We consider the difference between signal and value wires, and we extend our syntax and language to handle the connections between computation and wires. Finally, we present an example that includes control, data flow, and timing in one specification.

9.1 Value Wires

To illustrate the difference between signal and value wires, consider using a standard four-phase protocol on request and acknowledge wires to manage data transfer on a thirty-two bit bus. The request and acknowledge wires are signal wires; the edges, or value changes, are the events of interest. For instance, a rising edge on the request line indicates that a value is on the data bus and is ready to be transferred; a rising edge on acknowledge indicates that the data has been read off the data bus. Values of the data bus wires, on the other hand, are only valid between the rising edge on request and the rising edge on acknowledge. At all other times, the wires that comprise the bus may change arbitrarily. Thus, at first glance, we are primarily interested in the earliest and latest time that the data is on the bus as our events of interest. We call these events the `<` and `>` events, respectively; this is taken from the way the events are drawn on a timing diagram; figure 75 shows a simple orbital net and a timing diagram illustrating these events.

Care must be taken when modeling these events. Consider the case where we transfer two identical values over our four-phase interface, and the value on the bus just happens to remain constant over these two cycles. In this case, there is no physical event occurrence that would indicate when the first value was removed and the second value placed on the bus, since both values are identical.

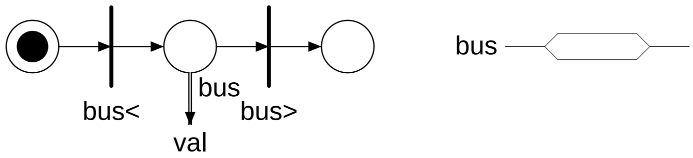


Figure 75. The events that occur on a value wire and the corresponding timing diagram. The data is valid during the thick part of the timing diagram.

Indeed, this is a general characteristic of value wires. There is no way for an observer to tell, by just examining the values of the individual wires of the bus, when a valid data item is on the bus. The control wires or some external clock indicate when the value wires contain valid data, not the bus itself.

9.2 Net and Language Syntax

For value wires we are primarily interested in the time during which a value is on a particular wire. Therefore, we associate values and wires with places, the only net type that represents duration. Indeed, the representation is natural; a data flow edge leaving a place models the value on a specific input wire during the time that a token exists in that place. A data flow edge entering a place models the requirement that a specific value is on an output value wire during the time that a token exists in that place.

To indicate which wire is of relevance for each of these arcs, we extend the syntax of orbital nets to include a function *place-wires* that returns, for each place in the net, a sequence of value wires. The *inpos* or *outpos* element of the data flow edge indicates which value wire in the list that data flow edge corresponds to. In illustrations, we write the name of the wire next to the data flow edge as it enters or leaves the place.

The primitive functions `value` and `setvalue` are used to obtain and specify the value on a particular value wire.

(`value` *wire*) For a given wire, models the value on that wire during the time that a token is in the current place. Returns a data flow edge connected to the current place.

(`setvalue` *wire expr*) Models that the value *expr* is on the wire *wire* when a token is in the current place. Evaluates the expression *expr* and connects a data flow edge from the result to the current place.

A net that uses four-phase handshaking and repeatedly sends the value one on its output is specified with

```
(def (ones (ack) (req bus))
   (process req+ (setvalue bus 1) ack+ req- ack-))
```

Similarly, a net that accumulates and sums the value that it receives on its input bus might be specified with

```
(def (sum (req bus) (ack))
   (let loop ((s 0))
      req+
      (let ((new-sum (+ s (value bus))))
         ack+ req- ack- ::
         (loop new-sum))))
```

Keys are useful for separating the interface specification from the data flow. Constructing an Orbits expression that evaluates into the desired net when describing the data flow timing requirements and at the same time sending the data flow value down the communications channel provided by a key can be cumbersome. The following two primitives simplify constructing such nets.

(`!-value` *key wire*) This primitive models the sending of the value on the specified input wire down the specified channel. This is implemented by annotating all input transitions to the current place with the appropriate `!` operation, using as input a data flow from the current place.

(`?-setvalue` *key wire*) This primitive models the setting of an output wire to the value received on the specified key. This is implemented by annotating all input transitions to the current place with the appropriate `?` operation and attaching the resulting data flow edge to the current place.

As an example, consider specifying our simple four-phase interface that sends the value down a channel. The expression

```
(process r+ (! k (value bus)) a+ r- a-)
```

does not produce the intended net; the transition created by the `!` statement is placed between the transition created by $r+$ and $a+$, and the place that is used to source the data flow edge then only reaches from $r+$ to the `!`. This is not what we want. The expression

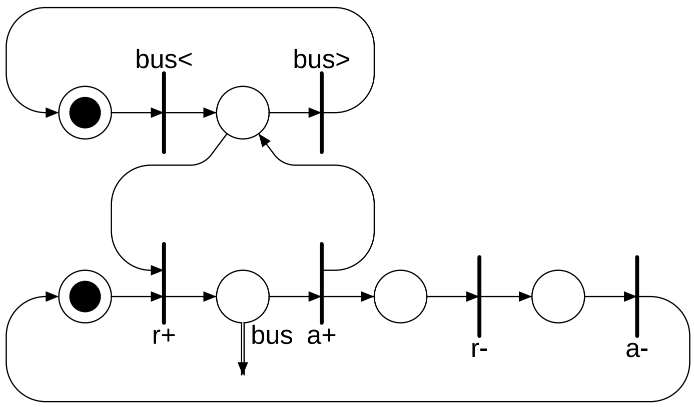


Figure 76. Enforcing the requirement that a value wire be valid and stable in order for its value to be used.

```
(process r+ (!-value k bus) a+ r- a-)
```

produces the desired four-transition net, with the `!` operation concurrent with the $r+$ operation.

A data flow edge coming out of a place acts similar to a join node, with the input transitions of the place firing an implicit register to store the value onto the node.

The `<` and `>` events are changes of value on the value wire, and are generated and used by models of data flow elements, but are not generally used in specifications. The connections between places and data flow elements, on the other hand, associate the value on a particular wire at a particular time with a data flow edge—these connections are used by both models and specifications. A token in a place with a data flow edge leaving or entering it implies that the value wire must be stable and valid, conditions met only in the interval between `<` events and `>` events. On the other hand, there can be any number of data items taken from a value wire during the time it is valid and stable. This relationship is enforced with a simple orbital net construction.

For each orbit that contains a place with a data flow edge into or out of it, such as the bottom portion of the net in figure 76, a small two-place net is constructed with a transition labeled with a `<` event and another transition labeled with a `>` event. A token in the place p between the `<` and the `>` event indicates that the wire contains a valid data item. All places p' that take as input or output a data flow edge on that wire have their input transitions extended to remove the token from place p and their output transitions extended to replace the token in place p.

9.3 Composition and Operational Semantics

In the current implementation of Orbits, we only support specification and simple simulation when using value wires, because we have not yet defined a satisfactory composition construction. Composition of `<` and `>` events can occur precisely as the corresponding `+` and `-` events on signal wires, but the construction is complicated by the data flow values that must be transmitted. We believe the most fruitful avenue for solving this problem is to expand the data flow and place connections into synchronized transitions and communication using keys, in which case the operational semantics requires no extensions.

Despite this unresolved issue, Orbits provides a powerful specification language. Consider describing a circuit about which we know the following:

- The circuit will compute the greatest common divisor of two inputs using Euclid's algorithm.
- The circuit will run in a loop, accepting input, calculating the results, and generating output. We do not initially specify whether accepting new input can be overlapped with computing the results or generating the output.
- Two input values will arrive in sequence on a particular data bus `din`. Each value will be valid on the rising edge of the wire `strobe`, with a setup time of at least three hundred nanoseconds and a hold time of at least one thousand nanoseconds.
- The `strobe` line will be high for a minimum of five hundred nanoseconds and will be low for a minimum of five hundred nanoseconds.
- The output must be placed on the data bus `dout`. When the output is ready and has been valid for at least three hundred nanoseconds, the output wire `stbout` must be raised. The output must then be held for at least one thousand nanoseconds.
- The `stbout` line must be high for a minimum of five hundred nanoseconds and must be low for a minimum of five hundred nanoseconds.

This circuit is described by

```
(def (mgcd-machine (stb din) (sto dout))
   (let ((kin (make-key 1)) (kout (make-key 1)))
      (par
         (process :500,inf: stb+ :500,inf: stb-)
         (process
            (par
               (!-value kin din)
               (begin :300,inf: stb+ :1000,inf:)))
         (process :500,inf: sto+ :500,inf: sto-)
         (process
            (par
               (?-setvalue kout dout)
               (begin :300,inf: sto+ :1000,inf:)))
         (process
            (let ((v1 (? kin)))
               (let mgcd ((a v1) (b (? kin)))
                  (if (= a b) (! kout a)
                     (if (< a b)
                        (mgcd a (- b a))
                        (mgcd (- a b) b))))))))))
```

Note the clean separation of the interface specifications and the data flow description. The resulting orbital net is given in figure 77.

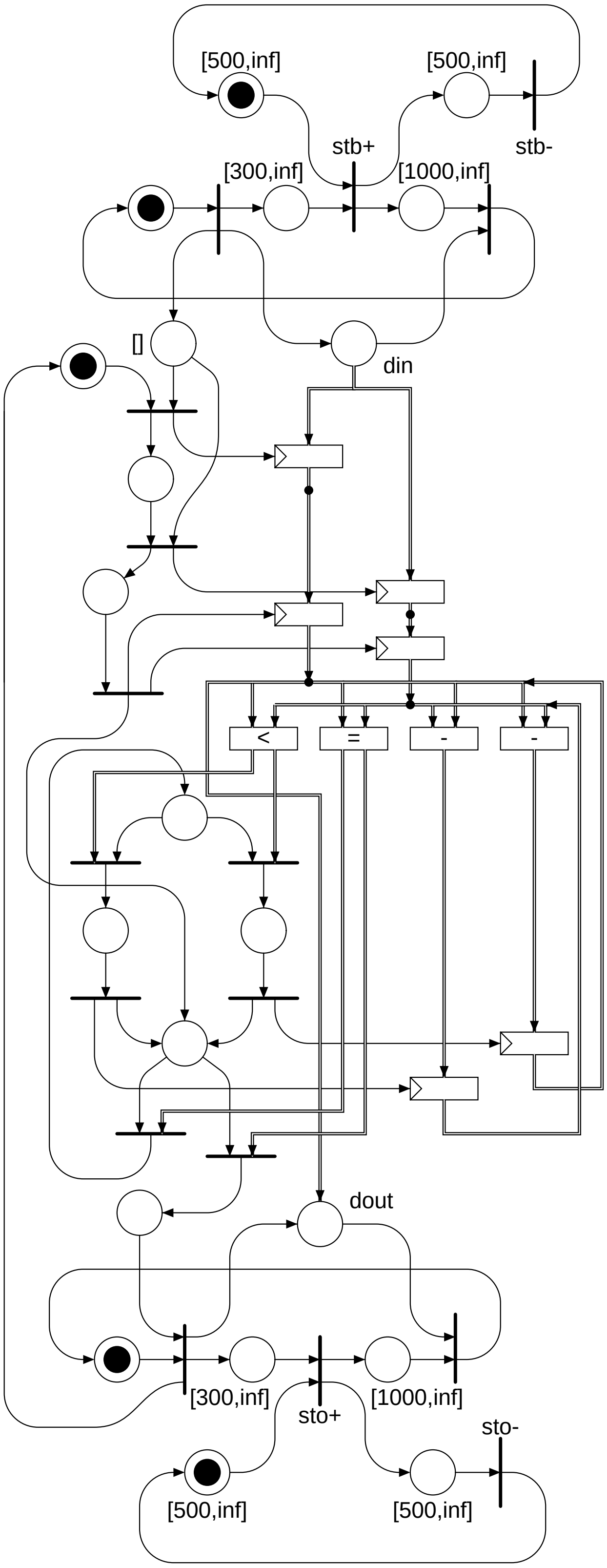


Figure 77. A complete orbital specification of a simple circuit.

Chapter 10

Future Directions

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning. —Winston Churchill, 1874–1965

This dissertation presented some solutions to the specification, modeling, simulation and verification of circuits. In particular, a rich representation was designed that describes control, timing, and data flow aspects of a circuit or specification. We presented a textual language that makes it much easier to use this internal representation. The representation has excellent operational semantics that allows quick and accurate simulation, and a particular behavioral semantics has been associated with the control and timing aspects of the representation that allows computationally efficient direct comparison of behavior. Essential timing algorithms and proofs have been presented that lead to much faster simulation and verification systems without sacrificing accuracy. Indeed, many of these latter algorithms and proofs are general and applicable outside of the Orbits formalism.

To prove the ideas presented, Orbits has been implemented as a computer program that, despite its experimental nature, provides excellent performance. Indeed, Orbits has been shown to be capable of handling systems with millions of states, orbital nets with tens of thousands of nodes, and timed circuits with greater accuracy and speed than other verification systems [11][28][1].

As is the case with much research, the questions posed by this work outnumber the answers supplied. In fact, so many exciting continuations of this research are possible that the main dilemma facing the author is which to undertake first. Though competitive and useful, Orbits is far from finished.

So much engineering has gone into the construction of Orbits that there is no way to describe it all. Specialized applications have required extensions such as a `burst` primitive that allows easy description of burst-mode specifications. The way arrays are stored, accessed, and shared in the timing algorithms provides a significant space savings. Various hash tables and look-aside tables are used to speed some common functions. As we gain experience with Orbits, such engineering enhancements will be commonplace.

Since Orbits provides a rich and expressive specification language, one of the most obvious applications is the integration of synthesis algorithms. Whether this is accomplished by translating the representation into another formalism that an existing synthesis system can handle, or whether Orbits is itself extended with synthesis algorithms, such integration would provide an excellent test of, and probably improvements to, the specification language. With Orbits providing modeling of circuit primitives, simulation, and verification, the result would be a complete high-level design system. Further integration of layout software would complete the picture.

Some less ambitious extensions can improve the usefulness of Orbits as it currently stands. A larger library of predefined elements should be constructed, and translations of other specification languages into Orbits should be undertaken, so that Orbits can provide simulation and verification services to existing CAD systems. Indeed, Orbits has already been integrated into one such synthesis system [38].

Many opportunities exist in the further improvement of simulation and verification performance. One technique that has not been investigated is the structural simplification of the orbital nets—some recent results [4] might yield a simpler net that admits faster simulation. Since orbital nets are essentially Petri nets, and since verification has been reduced to deadlock detection and liveness of certain transitions, recent work in unfoldings [31] shows great promise in reducing the exponential blowup of verification.

Liveness and fairness issues must be addressed to complete Orbits. While many liveness properties can be turned into real-time safety properties, this may not be as efficient as considering liveness in its own right. Orbital nets are timed transition systems, so model-checking techniques should be directly applicable. Perhaps orbital nets should be extended with accepting places analogous to Büchi accepting states in order to specify liveness requirements.

The timing algorithms can be improved even further. Different normalization techniques may have better performance than the one used in Orbits. Another interesting possibility is to take advantage of the discrete-continuous time behavioral equivalence of orbital nets to improve the worst-case performance by subtracting regions from one another and only exploring a region or regions large enough to include the previously unseen states. With this technique, the worst-case number of timed regions for a given system would be no worse than in the discrete case, and the average case performance might improve as well. Finally, rather than using simple timing intervals, a distribution might be associated with each interval to yield simulation runs that are distributed more properly over the expected behavior.

The operational and behavioral semantics of control and timing must be extended to data flow, particularly on wires. This would require a form of trace structures that admits data values, as well as composition constructions and an operational semantics for value wires. As a further improvement, symbolic data flow verification might yield excellent results, especially with the integration of a high-order logic proof system.

Finally, an intuitive graphical user interface, with schematic entry, net display and editing, and debugging facilities would make Orbits less of a chore to use. This may be possible through integration into an existing Petri net tool. Better simulation trace generation and statistical tools would allow performance evaluation with Orbits.

For those rainy days when progress in productive work proves elusive, Orbits can also pose some interesting riddles. For instance, the standard dining philosophers Petri net, with the philosophers taking the forks in a specific order, has an exponential number of states but only a single deadlock state. Random simulation (choosing a randomly enabled transition at each step) finds this deadlock state quickly—for two thousand philosophers, on average, only sixteen thousand simulation steps are required. Yet, verification requires an exploration of a substantial fraction of the state space before the deadlock is found—for fifteen philosophers, on average a half-million steps are required before the deadlock state is found. Why is there such a great disparity, since verification is effectively just exhaustive simulation? Because it is so easy to construct and experiment with simple models using Orbits, exploring these riddles is a pleasure, and such enjoyable investigations may yield further insights into the complex behavior of concurrent systems.

Bibliography

[1] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, August 1991.

[2] Tod Amon and Gaetano Borriello. On the specification of timing behavior. In *ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1990.

[3] Tod Amon, Gaetano Borriello, and Carlo Séquin. Operation/event graphs and Oesim. Technical report, Department of Computer Science, University of Washington, January 1990.

[4] C. Autant and Ph. Schnoebelen. Place bisimulations in Petri nets. In *Application and Theory of Petri Nets 1992*, pages 45–61, 1992.

[5] Peter A. Beerel, Jerry Burch, and Teresa H.-Y. Meng. Efficient verification of determinate speed-independent circuits. In *IEEE International Conference on Computer-Aided Design*, pages 261–267, 1993.

[6] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3), March 1991.

[7] Gaetano Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, University of California, Berkeley, May 1988.

[8] Gaetano Borriello and Ewald Detjens. High-level synthesis: Current status and future directions. In *25th Design Automatic Conference*, pages 477–482, 1988.

[9] M. C. Browne, E. M. Clarke, D. L. Dill, , and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computing*, C–35(12), 1986.

[10] J. A. Brzozowski and C.-J. H. Seger. Advances in asynchronous circuit theory; part I: Gate and unbounded inertial delay models. In *Bulletin of the European Association for Theoretical Computer Science*, volume 42, pages 198–249, October 1990.

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

[12] Steven M. Burns. *Automated Compilation of Concurrent Programs into Self-Timed Circuits*. PhD thesis, California Institute of Technology, December 1987.

[13] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, June 1987.

[14] David R. Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, 1989.

[15] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, Carnegie-Mellon University, 1988.

[16] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer-Verlag, 1989.

[17] Fintronic USA Inc. *UDL/I 0.1 Simulation Environment User's Guide*.

[18] Scott Hauck. Asynchronous design methodologies: An overview. Technical report, Department of Computer Science and Engineering, University of Washington, 1993.

[19] Thomas A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford University, August 1991.

[20] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[21] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Proceedings of the 5th Workshop on Computer-Aided Verification*, pages 3–14, 1993.

[22] Claude Jard and Thierry Jéron. Bounded-memory algorithms for verification on-the-fly. In *Proceedings of the 3rd Workshop on Computer-Aided Verification*, pages 192–202, 1991.

[23] David W. Knapp and Marianne Winslett. A formalization of correctness for linked representations of datapath hardware. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1989.

[24] Vadim Kotov. Personal communication.

[25] David C. Ku, Dave Filo, and Giovanni De Micheli. Control optimization based on resynchronization of operations. In *28th Design Automatic Conference*, pages 366–371, 1991.

[26] David Cho-Lun Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications.* PhD thesis, Stanford University, June 1991.

[27] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *28th Design Automatic Conference*, pages 302–308, 1991.

[28] Harry R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical report, Harvard University, July 1989.

[29] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the 1990 MIT Conference on Advance Research in VLSI*, pages 263–278, 1990.

[30] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 245–260. Computer Science Press, Inc., 1985.

[31] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the 4th Workshop on Computer-Aided Verification*, 1992.

[32] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[33] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt. Automatic synthesis of asynchronous circuits from high level specifications. *IEEE Transactions on CAD*, 8(11), November 1989.

[34] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communications*, COM–24(9), September 1976.

[35] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[36] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[37] B. Mishra. *Some Graph-Theoretic Issues in VLSI Design.* PhD thesis, Carnegie Mellon University, 1985.

[38] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. In *IEEE Transactions on VLSI Systems*, June 1993.

[39] J. A. Nestor. *Specification and Synthesis of Digital Systems with Interfaces*. PhD thesis, Carnegie-Mellon University, April 1987.

[40] Steven M. Nowick and David L. Dill. Practicality of state-machine verification of speed-independent circuits. In *IEEE International Conference on Computer-Aided Design*, pages 266–269, 1989.

[41] Steven Mark Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, March 1993.

[42] Zebo Peng. *A Formal Methodology for Automated Synthesis of VLSI Systems*. PhD thesis, Linköping University, 1987.

[43] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[44] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report 120, Massachusetts Institute of Technology, Project MAC, February 1974.

[45] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

[46] R. M. Shapiro. *Validation of a VLSI Chip Using Hierarchical Timed Petri Nets*, pages 667–690. Springer-Verlag, 1991.

[47] P. A. Subrahmanyam. TALES: Event-based semantics for timing specification (with applications to synthesis, verification, and analysis). In *ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1990.

[48] Texas Instruments Semiconductor Group. *The TTL Data Book for Design Engineers*. Texas Instruments, Inc., second edition, 1981.

[49] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn. The system architect's workbench. In *25th Design Automatic Conference*, pages 337–343, 1988.

[50] Antti Valmari. A stubborn attack on state explosion. In *Workshop on Computer–Aided Verification*, volume 1, 1990.

[51] W. M. P. van der Aalst. *Timed Coloured Petri Nets and their Application to Logistics*. PhD thesis, Eindhoven University of Technology, September 1992.

[52] Peter Vanbeckbergen. *Synthesis of Asynchronous Controllers from Graph-Theoretic Specifications*. PhD thesis, Katholieke Universiteit Leuven, September 1993.

[53] W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*. Springer-Verlag, 1992.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

REPRESENTING AND MODELING DIGITAL CIRCUITS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Tomas Gerhard Rokicki

December 1993

Abstract

A computer-aided circuit design tool can be described as a set of algorithms applied to an internal representation of a circuit design or specification in order to answer specific questions or perform certain operations. The representation directly limits the operations that can be performed as well as the accuracy of the resulting answers. This dissertation concerns the development and exploration of one such design representation and shows how it is appropriate for a wide range of circuit design tasks. Along the way, we discover some new techniques for specification, modeling, simulation, and verification of circuits. We especially consider real-time aspects of circuit behavior, and we present some significant enhancements to existing real-time verification algorithms.

We present a rich and expressive formalism, called *orbital nets*, that precisely describes control, timing, and data flow aspects of circuit behavior. We develop the concepts of synchronization, composition, and receptiveness for this formalism, and illustrate how it can be used to specify, model, simulate, and verify digital circuits.

We present a textual description language that provides convenient access to orbital nets through parameterization, partial evaluation, and structural and functional decomposition.

We present efficient hierarchical real-time simulation and verification algorithms. We prove the equivalence of continuous time and discrete time semantics for orbital nets, and develop new geometric timing algorithms that exploit concurrency to efficiently verify real-time safety properties.

Acknowledgments

This dissertation would not have been possible without the assistance and support of my thesis committee. Daniel Weise's enthusiasm, patience, and guidance helped me survive many years during which I wrestled with Orbits, and any good features in the final system are, in a large part, due to his patience and suggestions. David Dill supplied a tremendous amount of assistance in directing and focusing my research; without his direction I would still be adding features to Orbits with no dissertation in sight. I would also like to thank Giovanni De Micheli for his assistance and comments.

To Donald Knuth, I can only say thank you. Your books and research were my beacon to Stanford. I have never enjoyed anything at Stanford so much as being your teaching assistant in the problems seminar; I cannot believe that I will ever again see such a teacher and such a group of students. More than anything else, your example will always guide me in my research, my writing, and most importantly of all, my life.

I have been honored to work with Ludmila Cherkasova and Vadim Kotov over the last two years. Ludmila in particular has been a tremendously valuable sounding board for my ideas, pointing out flaws in my arguments, lack of rigor in my proofs, and counterexamples to my claims.

Of my other friends and colleagues at Stanford, I would like to thank in particular Martin Rinard and Elizabeth Wolf for their last-minute help, Peter Beerel, Jerry Burch, David Ku, Chris Myers, and Kenneth Yun for their discussions and suggestions, and Willy Langeveld, who has helped me keep my sanity over these past years. My wife, Susan Skulina, provided untold support and sacrificed countless hours of dancing and companionship; she has my gratitude and my love. The Fannie and John Hertz Foundation and the ARCS Foundation provided much appreciated financial support.

Table of Contents