The Specification and Execution of
Heterogeneous Synchronous Reactive Systems

Stephen Anthony Edwards

Please cite as

Stephen Anthony Edwards.
The Specification and Execution of Synchronous Reactive Systems.
PhD thesis, University of California, Berkeley, 1997.
Available as UCB/ERL M97/31.
`http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/`

Abstract

**The Specification and Execution of
Heterogeneous Synchronous Reactive Systems**

by
Stephen Anthony Edwards

Doctor of Philosophy in Engineering
University of California, Berkeley
Professor Edward A. Lee, Chair

The need for new languages and paradigms for designing software for embedded computing systems continues to grow as general-purpose microcontrollers become faster and cheaper. Many of these system need precise control over when things happen, yet few languages provide this facility. Another major challenge is handling the growing complexity of these systems.

In this dissertation, I present a new model of computation for embedded system software that is the first to fuse precise control over timing with the ability to build systems from heterogeneous pieces. It combines the synchronous model of time (used in languages such as Esterel) with the hierarchical heterogeneity of the Ptolemy system. Heterogeneity addresses the complexity problem by allowing each subsystem to be designed using the best language.

My two major contributions are the formal semantics of this model and an efficient, predictable execution scheme for it. Dealing with zero-delay feedback loops, a side-effect of the zero-delay assumption needed for synchrony, is the semantic challenge, and I solve it with a fixed-point scheme that guarantees all systems are deterministic by construction. The execution scheme I present is provably correct and

eliminates run-time scheduling overhead by making all decisions before the system is run.

I present results that show my model of computation is both efficient and can be used to implement practical systems. It is my hope that these ideas will be used in the future to make designing complex time-critical embedded software easier and less error-prone.

Contents

List of Definitions and Theorems

Preface

EMBEDDED computing systems are everywhere. If you own a VCR, a digital watch, a microwave oven, or an automobile, you probably use them daily without realizing it. This is a mark of good engineering: it solves a problem without calling attention to itself.

This thesis grew from a desire to simplify the task of building these systems. Their growing use of software makes it natural to attack the problem of creating software for these systems. Most of it is currently written using the C language, which was originally designed for operating systems programming. It is a powerful, flexible language, but was not designed for real-time programming where the correctness of a program rests as much on when it performs its function as on what function it performs.

The designers of digital logic (hardware) have long built such timing-critical systems, and their techniques have slowly been creeping into the software world. One of their most powerful paradigms is synchrony, where all parts of a system are synchronized to a periodic clock. Virtually all digital hardware systems use this, and it has recently entered the software world through a group of so-called synchronous languages that includes Esterel, Lustre, and Argos.

Another challenge in designing software systems is handling their complexity. Any reasonable scheme needs to address this, and the heterogeneous approach taken in the Ptolemy system (a system for designing embedded software systems) is one of the more interesting. The basic idea is to treat a system as a collection of black boxes. Within each black box there might be a program, a system, or anything. Carefully choosing the interface to these boxes allows systems built from them to be analyzed and executed without having to understand their contents.

PREFACE

The research presented here is the result of combining the idea of synchrony with the Ptolemy approach to heterogeneity. It presents a new model of computation (essentially, a way to assemble systems) that combines both of these ideas. The primary challenge, it turns out, is dealing with instantaneous feedback (the synchronous model is inherently instantaneous, and feedback appears in virtually all interesting systems). The solution I devised follows from results taken from both the programming language semantics community and the circuit simulation community, making it mathematically sound and based on physical principles.

The other big challenge with this approach is actually running the systems. The bulk of this thesis is devoted to making these systems run quickly, predictably, and correctly.

Some of the results in this work can be applied more widely. The problem of dealing with zero-delay feedback in software appears in the two major languages (VHDL and Verilog) currently used to specify hardware systems. Both have failings that could be corrected if some of the techniques presented in this dissertation were adopted. Also, the execution scheme I devised is essentially a very efficient solver for a system of equations. Although many of my techniques are closely tied to the particular domain I chose to work in, I believe the general approach is applicable to similar problems.

—Stephen Edwards
Emeryville, California
March 1997

Acknowledgements

Advisor:
Edward A. Lee

Adnan Aziz
Wendell Baker
Wan-Teh Chang
John Davis II
Jerry Edwards
Lois Edwards
Dan Engels
Brian Evans
Alain Girault
Michael Goodwin
Lisa Guerra
Nina Huang
Renu Mehra
Praveen Murthy
A. Richard Newton
Arlindo Oliveira
José Luis Pino
Rajeev Ranjan
H. John Reekie
Sanjay Sarma
Tom Shiple
Gitanjali Swamy

In my experience, most people read the acknowledgements section of a dissertation to either learn the name of the author's advisor or to see their name in print. For those in a rush, I have summarized this information in the easy-to-read list on the left.

For those still reading, here are more detailed comments:

Of course, I must begin with my advisor, Edward Lee, without whose help I would never have written this thesis. He gave me just enough direction and advice to get started in the right direction, then got out of my way. I must also thank Richard Newton, my previous advisor, who probably influenced this dissertation more than he or I realize.

Wendell Baker was instrumental in getting me started with synchronous languages in the first place. He also gave me a very useful piece of advice: "your objective should be to write as little code as possible." Eventually (it took a few years), I agreed with him and actually completed this work.

Gitanjali Swamy and her husband, Sanjay Sarma, were both great fun to have around. Gitanjali influenced my attitudes towards CAD research and towards cooking. Sanjay's advice on being an academic will influence me long after I have left Berkeley.

Dan Engels' caustic feedback on an early draft of this dissertation was invaluable. What you are reading was largely shaped by his cynicism. I can hardly wait to return the "favor" when he gets ready to write his dissertation. John Davis also read through an early draft.

Rajeev Ranjan and his wife Renu Mehra also deserve mention, not just because they fed me, but because Renu and her cubiclemate, Lisa

Guerra, were almost completely responsible for getting my qualifying exam presentation into shape.

Tom Shiple's discipline made me realize I still have a long way to go. Discussions with him about Malik's work and asynchronous circuit models greatly clarified my thinking on these matters.

Discussions with Adnan Aziz were a great help, even when they lead to him writing some unintelligible equation on the blackboard, something they always seemed to do. He also made me feel better about my own graduate student angst after explaining his story.

Arlindo Oliveira returned home to Portugal all too soon. He taught me a lot about how to approach research.

José Pino welcomed me warmly into Lee's group, and taught me a lot about the Ptolemy philosophy. He also made me realize that no matter how hard I though graduate school was, it was much harder with four children.

Discussions with John Reekie were always stimulating, assuming he had had enough coffee. Discussions with him about control and dataflow eventually lead to the scheme I present here, and he was a great help when I was building my synthesizer example.

Wan-Teh Chang was an excellent cubiclemate. He only seemed to open his mouth when he had a good suggestion about synchronous languages. Michael Goodwin took over his desk and gave me some good advice about the music synthesis world.

Brian Evans taught me a lot about being an academic. In addition to providing hair styling tips, he perfectly illustrated the attitude one needs to become a professor, inadvertently convincing me not to.

Praveen Murthy helped me work through some of the more sticky mathematical points in this dissertation. Without his help, I fear I would still be stuck.

Alain Girault reinforced my stereotypes of frenchmen. He's arrogant, dismissive, clever, and very helpful. I'll miss his flippant attitude and technical prowess.

Nina Huang was both my greatest inspiration and greatest distraction during this work. I thank her for both—her timing was perfect.

Of course, no acknowledgements section in a dissertation would be complete without some sappy comment about the author's parents, and unfortunately, this one will be no exception. I have a hard time imagining a set of parents who could be more supportive and helpful during such a challenging task. For this, I thank them both (Jerry and Lois Edwards).

Chapter 1

Introduction

Art is solving problems
that cannot be formulated
before they have been solved.
The shaping of the question
is part of the answer.
—Piet Hein

THE NEED for new languages and paradigms for designing embedded systems continues to grow. The falling cost of hardware has caused both the ubiquity and complexity of these application-specific computing systems to grow, and with more complexity comes a greater need to contain it. In this dissertation, I present a new model of computation—essentially a coordination language—for describing the software in these systems. It is the first to combine precise control over when things happen with the ability to assemble systems from pieces described in different languages, a way to fight complexity by allowing each piece to use the most suitable language.

My focus is on *reactive systems*,* systems that must respond to their environment at the environment's speed. When things happen in a reactive system is as important as what happens, making traditional computer programming languages insufficient because they only provide precise control of function. In contrast, my model of computation allows precise synchronization of events by assuming computation is infinitely fast. Familiar to designers of synchronous

*A term due to Harel and Pneuli [34].

digital logic, this divides time into a sequence of discrete "ticks" and allows the designer to control the tick in which an action takes place.

Software is becoming dominant in embedded system design because fast hardware is becoming cheaper. Earlier, custom hardware might have been required because of performance requirements, but now cheap, fast general-purpose microcontrollers are adequate for many of these jobs.

Fast, cheap hardware leads to greater system complexity since it allows larger, more powerful systems to be built. With complexity, however, comes the challenge of designing it correctly. Extensive simulation, and, currently to a lesser extent, formal verification can help in this process, but the easiest way to design a correct system is to design a simple system.

My Synchronous Reactive (SR) model of computation facilitates the design of simple systems because it can combine subsystems described in a variety of languages. For any particular problem, there is usually a language in which it can be solved elegantly. However, the variety of problems in a large system makes no one language ideal, so the need arises for a way to combine different languages. My model supports such heterogeneity by using coarse atomic units of computation: functions that can be as big as entire programs.

I made the SR model deterministic to simplify the design process. It is much more difficult to design and test a system with inherently unpredictable behavior[*] because both designers and analysis tools need to consider many more possible behaviors.

This model is the first to fuse the idea of instantaneous computation with support for heterogeneous system design. The primary challenge is to maintain the determinacy of such systems in the presence of zero-delay feedback loops. I discuss these problems informally in Section 1.3, and rigorously deal with the problem in Chap-

[*] This can occasionally be a good thing—nondeterminism is useful for modeling unpredictable environments.

ter 3, where I prove that my model of computation is deterministic.

A reasonable system description language should be defined formally, have a compilation procedure that produces efficient synthesized code (or, equivalently, have a very efficient simulation procedure), and be able to describe practical designs. A formal definition is necessary so that everything that manipulates the design, including the designer, can agree on what a design means. An elegant language that cannot be executed efficiently is not useful by itself, and an elegant language that cannot be used to describe anything useful is similarly useless.

My thesis is that my Synchronous Reactive model of computation is reasonable in this sense. In this dissertation, I present its formal definition and show it is consistent (Chapter 3), present an efficient way to execute it (Chapter 4), and exhibit a practical implementation along with some real examples (Chapter 5). In Chapter 2, I discuss some related system description languages and the final chapter is devoted to conclusions and speculation on future work.

1.1 Synchrony

Using digital circuitry to build logically correct systems has been extremely successful because it allows for abstraction. The idea is simple: using discrete values allows noise below a certain threshold to be filtered out *completely*. The result is an effectively noise-free circuit with behavior that is predictable and reproducible. This allows it to be treated as an ideal mathematical entity.

Synchronous circuits use the same idea to ensure temporal correctness. They discretize time to filter out "time noise" brought on by unpredictable, unmatched, and uncontrollable delays.

Synchronizing an outgoing event with an incoming one is the key ability here. Synchronous digial circuits generally have one synchronizing input: a periodic global clock signal. The synchronous model of time used in SR, which I adopted from the so-called synchronous languages,* is a generalization of this where every signal from the environment is effectively a clock. All output events are synchronized to the input events; none are produced without outside stimulus.

The ability to synchronize output events to *any* input event allows for great flexibility. In general, it is possible to make something happen on the *n*th occurrence of an event, such as on the tenth second (which requires a periodic "second" input), on the count of three, or on the fifth floor (e.g., for an elevator).

Concurrency is a fundamental requirement for synchrony. Traditional sequential languages such as C are not synchronous because they have no notion of concurrency. For things to be synchronized, they must happen simultaneously, yet a language like C is executed one statement at a time.

The synchronous model of time has a physical interpretation:

> **The Synchrony Hypothesis** The system computes infinitely quickly. Each reaction is instantaneous and atomic, dividing time into a sequence of discrete instants. A system's reaction to an input appears at the same instant as the input. (After Berry [5])

A system can behave synchronously if it is fast enough. Specifically, it must always finish its computations before more events arrive. Testing this amounts to testing the synchrony hypothesis, and requires knowing both the minimum inter-event time and the maximum computation time.

The synchronous model of time makes correct systems easier to design and build. It hides temporal details and simplifies the task of synchronizing parts of the system. Activity is easier to specify and understand because the behavior of the system is simplified. Moreover, the technique actually requires less control over the behavior of

*I discuss these in Section 2.3.

a system's components. Their exact speed does not matter provided it is above a certain threshold.

Unfortunately, it is not always practical to build synchronous systems. For example, physically distributed systems with long intra-system communication times are difficult to make synchronous. But for many applications, especially small embedded ones, synchrony makes sense.

1.2 Heterogeneity

Cheap hardware is enabling designers to create larger systems. These big systems are usually responsible for a wide variety of subtasks, such as a user interface, high-speed digital signal processing, communication, process control, and so forth.

Rarely is a single language ideal for describing each of these subtasks. A C program, for instance, is an excellent way to describe something like a database, but there are better alternatives for describing, say, signal processing. A poor choice of language—one far from the task or the implementation technology—often leads to an inefficient implementation, longer design time, and more design errors.

One approach is a "kitchen sink" language (such as the VHDL language [45], which includes behavioral and structural models), formed by forcibly combining a variety of computational models. Unfortunately, this is limited to using only those models included in the language and generally precludes later expansion. Moreover, analyzing systems described in such a language is harder because of the need to consider many models at once.

A more flexible alternative is to use a language that can coordinate the execution of and communication among subsystems described in a variety of languages. The challenge here is for the coordination language to cope with subsystems it does not understand completely. This approach can be summarized as follows.

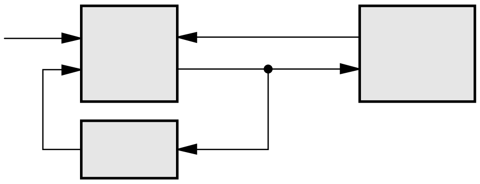


Figure 1.1 A simple SR system composed of three blocks communicating over three channels. The dangling channel on the left is an input from the environment. Some of the internal channels may be outputs to the environment.

> **The Black Box Approach to Heterogeneity** A system is treated as a set of "black boxes" whose contents may be arbitrary, but whose interfaces conform to a standard. A coordination language controls their execution and all communication between boxes.

When chosen correctly, a black box approach simplifies system analysis because it allows details such as the contents of the boxes to be safely ignored. By contrast analyzing a "kitchen sink" language is harder because the language is complex.

Unfortunately, the black box approach can prohibit the complete analysis of a system. When subsystems are treated too abstractly, certain properties about them cannot be determined. Unfortunately, the heterogeneous approach presented here precludes proving many correctness properties of systems. However, this is not necessarily a drawback because the systems in question are often so large that even if they were specified using a unified scheme, their analysis would be computationally intractable.

1.3 SR Systems

An SR system (one described using the SR model of computation) is composed of communicating blocks, as shown in Figure 1.1. The synchrony hypothesis assumes the inputs arrive as a sequence of discrete values and each block's computation is instantaneous. As a result, time in an SR system is a sequence of discrete "ticks," each initi-

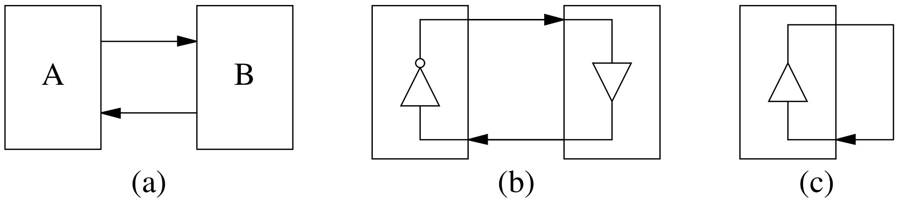


Figure 1.2 Problems with zero-delay and feedback. (a) Co-dependent: Which should be evaluated first? (b) Paradoxical: Appears to have no solution. (c) Ambiguous: Appears to have many solutions.

ated by the environment. In each tick, each block observes its inputs, instantly computes its outputs (which other blocks see in that same instant), and prepares itself (i.e., changes state) for the next tick.

Blocks communicate among themselves and with the environment through unbuffered unidirectional communication channels. In each tick, each channel takes on exactly one value; there is no buffering. Each channel is driven by either an output of some block or the environment, and may drive any number of block inputs. These connections, along with the number and type of all block inputs and outputs, do not change while the system is running.

1.3.1 Challenges of Zero Delay

Aside from the single-driver rule, no restrictions are placed on the topology of communication in SR systems. In particular, feedback, including self-loops, is permitted; some synchronous languages disallow them. Maintaining determinism (i.e., for each input there is exactly one reaction) with zero-delay blocks in the presence of feedback is the primary challenge in defining the behavior of these systems. Below, I describe the typical problems that arise in a zero-delay world and how I deal with them.

Ordering

To run the software system in Figure 1.2a, one of the blocks needs to be executed first. However, Block A depends on an output from Block B, so it needs to be evaluated later, yet Block B similarly depends on Block A, so which should be evaluated first?

I solve the ordering problem by separating the semantics of SR systems from their implementation. I treat an SR system as equations to be solved rather than as a sequence of functions to evaluate. Thus, it is the responsibility of the scheduler, not the designer, to ensure the blocks in such a system are evaluated in a sensible order. A scheduler based on the results of Chapter 4 might evaluate the blocks in the order ABA, but the designer has no control over this. Instead, the scheduler guarantees an order that produces a predictable result consistent with the formal semantics.

Paradoxes

The system in Figure 1.2b is paradoxical. The block on the left wants the two channels to take opposite values, yet the block on the right wants them to be equal, so what values should the channels take?

I solve such paradoxes by making "undefined" one of the possible values for the channels and restricting the class of functions the blocks may compute. The behavior of the system in Figure 1.2b is for both channels to be "undefined." This works because, for the block on the left, the opposite of undefined is undefined (this turns out to be the only reasonable choice), and for the block on the right, undefined is the same as undefined.

Nondeterminism

The system in Figure 1.2c appears to be ambiguous. The block only requires that its input and output take the same value, so it appears that the system may have any of a number of possible behaviors.

I deal with such ambiguity by choosing the least-defined solution. For Figure 1.2c, this means the channel will take the undefined value. Restricting the blocks to behave monotonically guarantees the least solution is unique. Moreover, this solution is the only one that does not require assumptions to be made about system behavior, making it more intuitive. The alternative would be a difficult-to-understand "guess-and-test" procedure that would form, test, and refine hypotheses about the values on each channel.

Here, monotonicity means a block will not recant or change its mind about a result. Given a more defined input, it will always produce a consistent output that may be more defined. Fortunately, any function that requires all its inputs to be defined before it produces any outputs is monotonic, making it easy to embed an arbitrary function in an SR system. Many familiar imperative languages (e.g., C, C++, and most assembly languages) implicitly compute such strict functions, so importing blocks from such languages is straightforward.

Chapter 3 is devoted to an extensive, rigorous discussion of the semantics of SR systems, including precise definitions of monotonicity, "undefined," least solutions, and the like.

1.3.2 Execution

My execution procedure for SR systems* is based on the idea of relaxation. I calculate the behavior of the system by repeatedly choosing and evaluating blocks until the system has converged to where no block would change the value on any channel. Requiring the blocks to behave monotonically ensures this procedure will always terminate with a unique result. The convergence time is bounded since each channel may become defined at most once in an instant, and there are a fixed, finite number of channels. It can be shown that the

*Others are possible since the semantics in Chapter 3 say nothing about the execution procedure.

result is the same regardless of which blocks are chosen.

Carefully choosing the block evaluation order makes this execution scheme efficient and predictable. Results in Section 4.3.6 show that in practice the worst-case execution time grows slower than $n^{1.5}$, where n is the number of block outputs. The challenge is dealing with feedback loops, which I do through a recursive divide-and-conquer strategy that systematically breaks certain feedback loops, iterating them to convergence. Although others have taken the same general approach, mine is the only one that is provably optimum. All of this is discussed in great detail in Chapter 4.

For a block to work within an SR system, it must have an SR interface and be able to compute a monotonic function of its inputs on demand. Beyond that, the "guts" of an SR block can be described and implemented in any way, allowing for heterogeneity. The algorithms in Chapter 4 only needs to know the communication structure of the blocks; not their contents.

A useful side-effect of the heterogeneity of SR systems is their support of truly hierarchical designs. Any group of SR blocks can be encapsulated in a single block without affecting the behavior of the system (although this sometimes affects performance), allowing subsystems to be compiled separately. Currently, all other synchronous languages are "flattened" before they are executed, prohibiting separable compilation and limiting the size and complexity of designs.

Chapter 2

Specification

A specification that will not fit
on one page of 8.5 × 11 inch paper
cannot be understood.
—Mark Ardis

ANY SYNCHRONOUS system with bounded resources behaves like a finite-state machine (FSM), a well-understood and conceptually simple entity, yet in practice such a system is rarely described as an FSM. In this chapter, I discuss why this is and argue the need for a coordination language such as SR to combine subsystems described using application-specific languages. I also discuss some of the languages that inspired SR.

2.1 Synchrony and Finite-State Machines

A synchronous system with bounded resources behaves like a finite-state machine. In each instant, the system receives a block of input and produces a block of output based on it. The behavior of such a system is usually time-varying, meaning the output in an instant is a function of both the input in that particular instant and the *history* of the system—the inputs in all earlier instants. The history of the system can be thought of as its *state*—an internal configuration that affects the output function and changes from instant to instant. Bounded memory resources can only distinguish a finite number of these histories, hence the machine has a finite number of states.

A synchronous finite-state machine consists of is six things:*

$$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$$

where

- Q is the finite set of states
- Σ is the finite input alphabet: a set of symbols
- Δ is the finite output alphabet: a set of symbols
- δ is the transition function mapping $Q \times \Sigma$ to Q
- λ is the output function† mapping $Q \times \Sigma$ to Δ
- q_0 is the initial state (in Q)

When a state machine is in state q and input a arrives, it produces output $\lambda(q,a) \in \Delta$ and goes to state $\delta(q,a) \in Q$. It starts in state q_0.

This is the complete story for *every* synchronous system. In theory, only these six things need to be described; in practice, each can easily become unmanageably complex.

Describing the input and output alphabets is often the easiest task. Sometimes they are small enough to be listed directly, or they may be a simple subset of a familiar set such as the integers. More often, they are sets of vectors described using a complex data type from a programming language such as C or Pascal.

In contrast, describing the state set, the transition function, and the output function is difficult because of the sheer size of the domains involved. A vector-valued input alphabet grows exponentially with the width of its vectors, so even small vectors can render an enumeration-based description of the output or transition functions

*This notation is taken from Hopcroft and Ullman [35], a standard reference on the subject of automata theory.

†Note that for reactivity, the output depends on both the state and the input.

impractical. Clearly, a useful specification scheme must allow a designer to succinctly specify exponentially large sections of these. In general, this is the problem of succinctness.

2.2 Succinctness

Succinct description is a goal of all languages. It is generally easier to make a short description correct because there are fewer places to make mistakes. Similarly, analyzing a succinct description is usually easier because solving a small problem is usually easier than solving a large one.

An ideal design language would allow succinct descriptions of *all* designs, but this is theoretically impossible because there are simply too many possible designs. Real languages try instead to make the description of some reasonable subset of designs succinct; other systems have either a verbose description or none at all.

This fundamental barrier is partly responsible for the enormous number of design languages that have been developed. Designers and design tools alike crave succinct descriptions, so many application areas have had special-purpose languages designed for them.

As systems grow larger and more diverse, however, it becomes less likely that a single language will be able to succinctly describe all parts of a given system. Although for each subsystem, there will be a language that can describe it succinctly, no one language will be the best for all subsystems.

One solution to this problem is the ability to connect and coordinate heterogeneously-specified subsystems. In this way, existing work on specification languages can be leveraged to provide more powerful ways to specify systems. This is the heterogeneous philosophy behind SR.

In this thesis, I concentrate on one way of combining subsystems (i.e., concurrently) that appears to be a very natural way for designers to think. It can be found in virtually all higher-level languages for de-

> The system has two inputs, *reset* and *next*, and three outputs, a, b, and c. Whenever *reset* appears, a is emitted. After this, the first *next* signal produces a b, and the second *next* signal produces a c.

Figure 2.1 The sequencer example, a simple reactive system, described in English.

scribing reactive systems (e.g., those presented in the next section), and is often the hardest aspect of these languages to design correctly because of the sometimes paradoxical implications of zero delay.

2.3 Synchronous Languages

In this section, I present a collection of synchronous languages* that illustrate some of the issues that arise in specifying synchronous systems. All rely on the synchrony hypothesis, and all are capable of specifying arbitrary finite-state machines, yet for a particular design, one is usually better than the others. To contrast the languages and illustrate this point, I have implemented a simple reactive system in each language, described in Figure 2.1 and hereafter called the sequencer example.

The description in Figure 2.1 is deliberately vague to illustrate a point. Especially in synchronous designs, it is easy to overlook a particular case, yet the system must handle all cases. When *next* and *reset* appear together, what should happen? The description suggests a is emitted, but what about the other outputs? I have chosen to make a take precedence, but other choices are possible. The right one usually depends on the system's environment.

The languages I present in this section range from the obvious to the subtle. The most obvious lists the output and next-state functions

*Halbwachs's book [32] and a special issue of Proceedings of the IEEE [3] provide a more comprehensive summary of these.

in a table. Traditional state diagrams are essentially these tables with a graphical syntax and input predicates. Derived from these are the textual OC format, which introduces more sophisticated predicates and actions, and the graphical Argos, which just adds hierarchy and concurrency. The imperative language Esterel departs completely from an explicit list of states. Lustre is an even greater departure, concentrating almost exclusively on arithmetic and having very little notion of state.

Each language needs some procedure for checking the validity of a description. The difficulty of this varies with the language and the level of validity to be verified, but in general the more succinct the descriptions, the harder it is. This is unfortunate, but is a natural side-effect of languages that allow a succinct description of complex behavior.

2.3.1 Tabular Form

The most obvious way to describe a synchronous finite-state machine is to list the output and next state functions for each possible input and present state, e.g., Figure 2.2. Even such a small system illustrates the problem with this approach—the number of rows in the table grows exponentially with the number of inputs.

Checking that a table is consistent is simple: there must be exactly one row for each state/input combination, and each output and next state must be an allowed output or state.

2.3.2 State Diagrams

State diagrams (e.g., Figure 2.3) are a slight improvement over tables. These are graphs where each node represents a state. Each arc is labeled with an input that causes a transition from one state to another and the output produced when this happens. The labels take the form “input predicate/outputs.” The input predicate is a conjunction of true and complemented signals, and the outputs are simply a list of

reset	next	PS	NS	a	b	c
0	0	A	A	0	0	0
0	1	A	A	0	0	0
1	0	A	B	1	0	0
1	1	A	B	1	0	0
0	0	B	B	0	0	0
0	1	B	C	0	1	0
1	0	B	B	1	0	0
1	1	B	B	1	0	0
0	0	C	C	0	0	0
0	1	C	A	0	0	1
1	0	C	A	1	0	0
1	1	C	A	1	0	0

Figure 2.2 The sequencer example described with a table. Inputs are on the left; outputs are on the right. PS is "present state;" NS is "Next State."

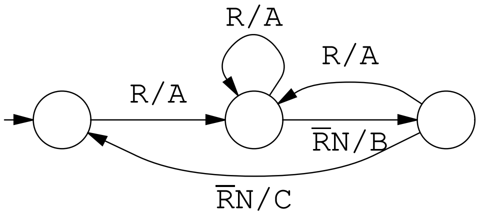


Figure 2.3 The sequencer example described using a state diagram

signals to emit. By convention, if no arc from the current state has a true predicate, the machine stays in that state and emits nothing. The initial state is marked with a short arrow leading to it.

State diagrams are usually preferable to tables, since the multi-dimensional nature of the transitions are more easily visualized, but they are not much more compact. The only advantage comes when the predicates are simple relative to the number of cases, or when most actions are "do nothing."

Using predicates instead of an explicit representation makes consistency checking more difficult. A state diagram is nondeterministic if there are ever two arcs from a single state with simultaneously true predicates, something that requires knowing all possible inputs.

2.3.3 The OC Format

The OC ("Object Code") format [68] (see also Caspi et al. [19]) was developed as a common intermediate language for the synchronous languages Esterel, Lustre, and Argos. Of the synchronous languages I present in this section, OC code is the easiest to execute on a sequential processor. It is well-suited to describing sequential control processes, but does not have any of the concurrency or preemption of some higher-level languages.

An OC program describes a single finite-state machine. Each state has an attached decision tree whose nodes are indices into an action table and whose leaves are pointers to next states. The action table is a list of atomic behaviors that include testing a signal or variable, emitting a signal, computing the new value of a variable, or calling an external function.

Figure 2.4 depicts the sequencer example described in a stylized OC format. For such a simple example, it is comparatively verbose, but it allows much more complex predicates and actions, including arithmetic.

Representing concurrent behavior with an OC program is difficult

```
Action Table:
0: if R then
1: if N then
2: emit A
3: emit B
4: emit C

State 0:          State 1:          State 2:
(0) if R then     (0) if R then     (0) if R then
(2)   emit A      (2)   emit A      (2)   emit A
      goto 1            goto 1            goto 1
    goto 0        (1) if N then     (1) if N then
                  (3)   emit B      (4)   emit C
                        goto 2            goto 0
                      goto 1            goto 2
```

Figure 2.4 The sequencer example described in OC

because it only describes a single FSM. For example, in the Esterel program of Figure 2.5a, it is fairly easy to see that signal B only depends on signal A, yet if the program is compiled into OC, B may also appear to depend on C in the resulting program (Figure 2.5b), incorrectly causing the system in Figure 2.6a to deadlock. There are other ways to compile this program, but all suffer from this problem of artificial dependencies. The problem, fundamentally, is that OC forces two events that could happen simultaneously to happen in a particular order.

2.3.4 Argos

Maraninchi's hierarchical finite-state machine language Argos [48, 49, 30] is a purely synchronous derivative of Harel's informal but influential Statecharts [33]. Later attempts to formalize the Statecharts semantics [56, 36] were somewhat successful, but the confusion has

```
module TWOWIRES:
input A, C;
output B, D;
  every A do emit B end
||
  every C do emit D end
end module
```

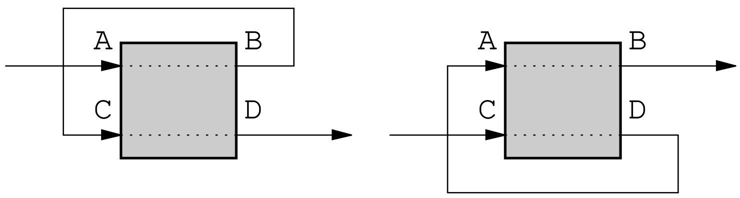
(a)

```
Actions:
0: if C then
1: if A then
2: emit B
3: emit D

State 0:
(0) if C then
(3)      emit D
(1)      if A then
(2)          emit B
         goto 0
(1)  if A then
(2)      emit B
    goto 0
```

(b)

Figure 2.5 (a) A simple Esterel program. (b) Its OC representation. Parentheses surround action numbers.



(a) (b)

Figure 2.6 The problem with flattening concurrency. If the order in which the module processes its inputs is fixed, one of these systems will incorrectly deadlock.

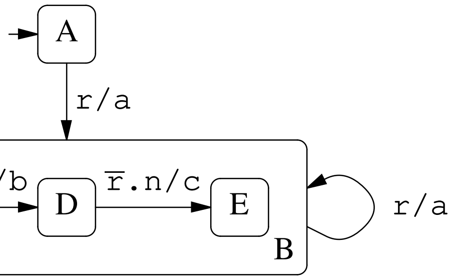


Figure 2.7 The sequencer example described in Argos

resulted in some twenty-two variants of the language [72], of which Argos is perhaps the cleanest.

An Argos specification is a hierarchical state diagram. When a state encloses one or more states, there are two possibilities. If the inner states are "OR" states (e.g., those in State B in Figure 2.7), exactly one of the inner states is active when the enclosing state is active; if the inner states are "AND" states (drawn with dashed lines separating them, as in Figure 2.8), *all* of the inner states will be active.

Figure 2.7 shows an Argos implementation of the sequencer example. It starts in State A and waits for the `r` signal. When `r` is present, the system emits the `a` signal and enters States B and C, since State C is the initial state (denoted by its sourceless arrow) in the collection of OR-states in State B.

The Argos semantics require the ability to partially evaluate input predicates. Figure 2.8 illustrates this. When the signal `x` is present, neither $\texttt{x.}\overline{\texttt{y}}$ nor `x.y` holds since the status of `y` has not been established, but `a` is emitted anyway since it is the action in both cases. This allows the self-loop on State C to fire, emitting `y`, and causing the arc to State B to fire completely.

Checking an Argos program for consistency is more difficult than checking a state diagram. Again, determinism requires that no more than one arc from a state have a true predicate. This requires at least some boolean analysis, but a more precise check might take into ac-

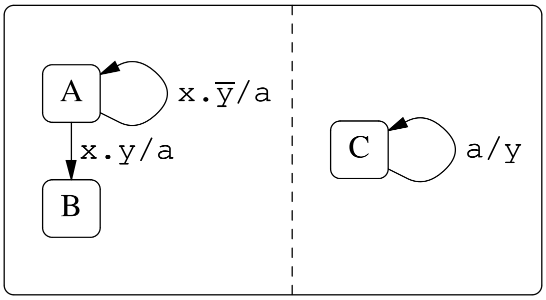


Figure 2.8 Incremental behavior in an Argos program. When in State A, x arriving causes a and y to be emitted and the system to enter State B.

```
module RESTART:
input RESET, NEXT;
output A, B, C;

  every RESET do
    emit A ;
    await NEXT ; emit B ;
    await NEXT ; emit C
  end

end module
```

Figure 2.9 The sequencer example described in Esterel

count which states (actually, combinations of states) of the system can be reached through some sequence of inputs. Solving this problem for large systems is currently at the frontier of formal verification research.

2.3.5 Esterel

Berry's synchronous language Esterel [7, 4] is textual, imperative, and well-suited for specifying sequential control-dominated tasks. It is concurrent and deterministic, and supports preemption and exceptions. An Esterel program is a group of concurrently-executing mod-

Statement	Meaning
`nothing`	Do nothing.
`pause`	Pause until the next instant.
`signal` S `in` s_1 `end`	Introduce local signal S and execute s_1.
`emit` S	Make S present in this instant.
s_1 `;` s_2	Execute s_1. When it terminates, execute s_2.
s_1 `||` s_2	Execute s_1 and s_2 until both terminate.
`loop` s_1 `end`	Execute s_1 and restart it when it terminates.
`present` S `then` s_1 `else` s_2	If S is present, execute s_1, otherwise execute s_2.
`suspend` s_1 `when` S	Execute s_1 in the current instant and in later instants where S is absent.
`trap` E `in` s_1 `end`	Introduce the local exception E and execute s_1.
`exit` E	Terminate the enclosing `trap` E statement.

Table 2.1 Esterel kernel statements. s_1 and s_2 are statements, S is a signal name, and E is an exception name.

ules that communicate through signals that in each instant are either absent or present with a value.

Figure 2.9 shows Esterel can be very succinct in specifying sequential behavior. Essentially, a three-state machine that emits A, B, and C is enclosed by a loop that restarts it whenever reset appears.

The language consists of a set of kernel statements from which other, more complex control structures are built. This kernel,* which deals only with pure (non-valued) signals, is shown in Table 2.1. For example, the derived statement `await` S, which terminates in the next instant in which S is present, can be built from kernel statements as follows:

*The kernel has continued to evolve since its first incarnation. The kernel presented here is from Berry's book [6].

```
trap T in
  loop
    pause ;
    present S then exit T end
  end
end
```

The full Esterel language also has simple arithmetic operations and variables along with a host of higher-level control constructs built from kernel statements.

Reincarnation is an odd aspect of the Esterel language. In certain cases, such as the one below, a signal may appear to take two or more values in a single instant.

```
loop
  signal S in
    present S then emit O else nothing end ;
    pause ;
    emit S
  end signal
end loop
```

In the second instant, the signal S is emitted, the `signal` statement terminates, and the loop resets with a fresh, absent copy of the S signal. Signal O is not emitted. Detecting these cases and correctly expanding them into a format like OC has been a challenge for those writing compilers for the language.

Checking the consistency of an Esterel program is more difficult than any of the other languages presented here. It is easy to write paradoxes (see Section 1.3.1) in the language, and exactly checking for them involves exploring every possible execution of the program. The latest compiler (V4, as of this writing) does this symbolically after converting the program to a circuit. See Section 3.1.2 and Shiple et al. [64] for more details.

Constants	`true 0 -5 5.3e-2`
Variables	`a x is_set`
Arithmetic operators	`+ - * / div mod`
Boolean operators	`and or not`
Relational operators	`= < <= > >=`
Conditional	`if then else`
Delay	`pre`
Initialization	`->`
Downsampling	`when`
Upsampling	`current`

Table 2.2 Components of Lustre flow expressions.

2.3.6 Lustre

Caspi et al.'s Lustre language [20, 31] is a declarative, textual synchronous language with a dataflow flavor. A Lustre program consists mainly of expressions that define *flows*—a possibly infinite sequence of values of a particular type along with a clock, a sequence of times for the sequence of values. All of these expressions are running concurrently and are order-independent.

Lustre flow expressions are built from the components shown in Table 2.2. Operators work pointwise on flows with identical clocks, a compiler-enforced restriction. For example, if `x` and `y` are flows with values $(x_1, x_2, \ldots)$ and $(y_1, y_2, \ldots)$ and identical clocks, then `x+y` $= (x_1 + y_1, x_2 + y_2, \ldots)$.

Delay and initialization operators add sequential behavior to the language. The `pre` operator adds memory—it delays a flow by one clock cycle. Specifically, `pre x` $= (nil, x_1, x_2, \ldots)$ (*nil* denotes undefined). The `->` ("followed by") operator makes it possible to initialize memory by changing the first value of a flow. Specifically, `x -> y` $= (x_1, y_2, y_3, \ldots)$.

`C`	0	1	0	1	0	0	1	0
`X`	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
`Y = X when C`		x_2		x_4			x_7	
`current Y`	*nil*	x_2	x_2	x_4	x_4	x_4	x_7	x_7

Figure 2.10 The relationship between the `when` and `current` operators.

```
node RESTART(reset, next: bool)
  returns (a, b, c: bool);
var clock: bool;
let
  clock = reset or next;
  (a, b, c) =
     current( COUNTABC(reset when clock) )
       and clock;
tel.

node COUNTABC(reset: bool)
  return a, b, c: bool;
let
  a = reset;
  b = (false -> pre(a)) and not reset;
  c = (false -> pre(b)) and not reset;
tel.
```

Figure 2.11 The sequencer example described in Lustre

Two sampling operators impose a tree structure to the clocks in a Lustre program. The downsampling `when` operator creates a flow whose clock is defined by a boolean flow; the upsampling `current` operator interpolates a flow so that its clock is the one on the boolean flow that generated the clock. Figure 2.10 illustrates the relationship between these two operators. The compiler uses a simple syntactic unification algorithm to tell when clocks on signals are identical.

Figure 2.11 shows the sequencer example written in Lustre. The specification is clumsy because the example is sequential—Lustre is better-suited to specifying multirate dataflow systems.

Consistency checking is fairly easy for Lustre. Feedback loops without a `pre` operator are prohibited, something easily checked. The other challenge is checking clock consistency, which amounts to verifying the clocks on the signals feeding to an operator are the same.

2.4 Heterogeneous Languages

In this section, I review two languages supporting heterogeneity that inspired my own. Unlike SR, both are targeted toward data-centric applications, but they illustrate the heterogeneous approach to system specification.

In Kahn's programming language, the restrictions on the interface and contents of the blocks ensure determinacy. Further restrictions give Lee's Synchronous Data Flow, which trades some of the flexibility of Kahn's scheme for nearly complete compile-time analysis, including memory usage, termination, and run-time behavior.

2.4.1 Kahn Process Networks

Kahn, in an early influential paper [38], presented a simple language for parallel programming based on a process model. It defines a system as a set of parallel-executing processes that communicate exclusively through single-input, single-output FIFOs. When a process reads a data token from one of these FIFOs, it blocks until one is available. Kahn showed that this restriction was sufficient to make these systems determinate, rendering the sequence of data tokens on each FIFO independent of process execution order or speed.

Figure 2.12 shows a simple process in Kahn's language that acts as a switch. Integers on input `U` are alternately sent through outputs `V` and `W`. As its name suggests, the `wait` statement waits for the next value to arrive on an input.

Executing these networks without doing unnecessary work or using more memory than needed is challenging. Compile-time analysis is impossible in general, since each process can be described in a

```
Process g(integer in U; integer out V, W);
  Begin integer I; logical B;
    B := true;
    Repeat Begin
      I := wait(U);
      if B then send I on V else send I on W;
      B := not B;
    End;
  End;
```

Figure 2.12 A process in Kahn's language.

Turing-complete language. Kahn and MacQueen [39] discussed this problem in a later paper, and Parks [54] solves the problem with a scheduling scheme that runs one of these networks in bounded memory and time if possible.

2.4.2 Synchronous Data Flow

Lee and Messerschmitt's Synchronous Data Flow (SDF) [43, 42] is another block diagram language that takes a heterogeneous approach. It is well-suited for describing multirate digital signal processing systems and can be compiled to produce very efficient, predictable code. Figure 2.13 shows a typical SDF application—a modem.

SDF is a subclass of the class of dataflow process network languages,* which are themselves a subclass of the Kahn process network languages. SDF gives up Turing-completeness in return for extensive compile-time predictability. In particular, it can be scheduled statically, removing all run-time scheduling decisions and allowing memory consumption to be predicted exactly.

An SDF system is composed of a collection of blocks that communicate through single-driver, single-receiver FIFO buffers. The exe-

*see Lee and Parks [44] for a good summary of these cution of a block is divided into atomic “firings” where the block consumes and produces a fixed number of tokens on each FIFO. In executing the system, all blocks are fired in a sequence that periodically returns the number of tokens on each buffer to its initial value. In this way, the system can run forever without over- or under-flowing any communication buffer.

Knowing such a sequence of block firings at compile time leads to a simple compilation technique known as block code generation. In it, the code for each block’s firing is concatenated together according to the firing sequence. The advantage of this is that the code for each block firing can be optimized by hand—a boon for programmable DSPs, whose code is often difficult to optimize automatically.

An SDF system is deterministic because it is a Kahn process network. The sequence of tokens that appear on each FIFO is guaranteed to be the same for any valid execution of the system.

One problem with SDF is that it is not compositional. Coalescing two blocks into one can cause deadlock where none would exist in the original system. This does not always happen, and there are heuristics for avoiding it (see Bhattacharyya et al.’s work on scheduling SDF graphs [8]), but it cannot be avoided in general. It is disturbing, however, that something as simple as two wires cannot be modeled as an SDF block. This diminishes the heterogeneous nature of SDF, implying there is a class of subsystems whose behavior cannot be encapsulated. Furthermore, designs cannot be truly hierarchical; all hierarchy must be flattened completely to avoid introducing artifical deadlock.

SR avoids SDF’s compositionality problem by effectively allowing partial or incremental firing of blocks. It is SDF’s inability to do this that prevents it from modeling a wire. SDF’s firing rules impose more synchronization at block boundaries than is present in the rest of the system. Since SR systems are completely synchronous, this problem does not arise.

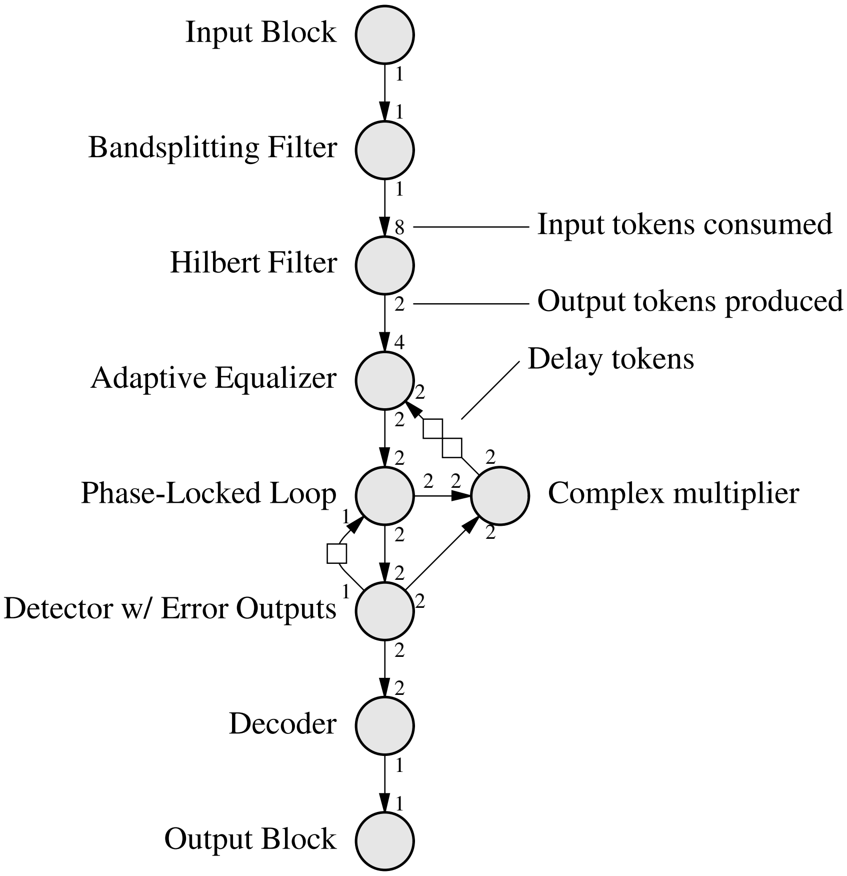


Figure 2.13 An SDF description of a modem, adapted from Lee [41].

Scheme	Blocks	Communication	Concurrency
SR	Monotonic functions selected by a state machine	Single-driver, multiple-receiver channels, one value per instant	Each block computes a function in each instant and advances its state
Argos	Hierarchically-nested finite-state machines	Broadcast signals present or absent in each instant	Each FSM produces output and advances its state each instant
Esterel	Sequential and parallel imperative statements	Broadcast signals present or absent in each instant	In each instant, each block runs until it is waiting for the next instant
Lustre	Arithmetic & boolean operators, delays, down- and up-samplers	Broadcast flows: sequences of data with an associated clock	Each operator computes once each instant
Kahn	Sequential imperative statements	Unbounded unidirectional FIFOs with blocking reads	Each process runs unless blocked by a read from an empty FIFO
SDF	Produce and consume a fixed number of tokens each firing	Unidirectional FIFOs; size computable at compile time	Processes fire in a repeating sequence

Table 2.3 A comparison of some system specification schemes

Chapter 3

Semantics

The test of a first-rate intelligence
is the ability to hold two opposed ideas
in the mind at the same time,
and still retain the ability to function.
—F. Scott Fitzgerald

IN THIS CHAPTER, I formally define the semantics of the SR model of computation. The main problem is unambiguously interpreting systems with zero-delay feedback loops, which I do by treating the blocks of a system as a system of equations. I use a well-known theorem from discrete mathematics to show the system has exactly one solution. To my knowledge, these are the first formal semantics for heterogeneous synchronous systems.

The meaning of an SR system in an instant is the least solution of $f(x) = x$, where x is the values in the communication channels and f is the function computed by the blocks for a particular set of inputs. By restricting the blocks' behavior to be monotonic and making certain values of x more defined than others, there is always a unique least x for any input, making SR systems deterministic.

I take a fixed point approach because both the programming language semantics community and the digital circuit simulation community use it to give meaning to recursive or self-referential entities. As such, it is both mathematically sound and physically realistic.

The semantics I present say nothing about how to execute these systems. This was deliberate—by not addressing the problem, it becomes easier to devise new ways to execute these systems. For example, a scheduler for simulation might minimize average execution time, whereas a scheduler for implementation might minimize worst-case execution time. Any approach whose result adheres to the semantics is acceptable.

This chapter contains three sections. In the first, I further motivate the fixed-point approach by reviewing the approaches two communities have taken to similar problems. In the second, I review the discrete mathematics of complete partial orders and continuous functions, which I use in the third section to define the semantics of SR Systems, ultimately showing they are unambiguous and thus deterministic.

3.1 Motivation

It is surprising that both the programming language semantics and the digital circuit simulation community arrived at nearly the same solution to assigning meaning to recursive or self-referential entities. After all, the simulation community had to choose something that matched physical reality, whereas the semantics community was free to choose any mathematically sound approach. Despite these differences, the solution is roughly the same, suggesting it is somehow natural.

In the remainder of this section, I present the solution to this problem from each community's viewpoint. The core ideas form the basis for the formal semantics I present in Section 3.3.

3.1.1 Denotational Semantics

In the denotational approach to programming language semantics, pioneered by Dana Scott and Christopher Strachey [62, 61, 65] in the early 1970s, the meaning of a program fragment is defined by mapping it to an element, often a function, in a semantic domain. For example, the recursively-defined factorial function in Figure 3.1a might

```
int fact(int x) {
  if ( x == 0 ) return 1;
  else return x * fact(x-1);
}
```

(a)

$$
\begin{array}{rcl}
 & \vdots & \\
f(-2) & = & \perp \\
f(-1) & = & \perp \\
f(0) & = & 1 \\
f(1) & = & 1 \\
f(2) & = & 2 \\
f(3) & = & 6 \\
f(4) & = & 24 \\
 & \vdots &
\end{array}
$$

(b)

Figure 3.1 (a) A recursive definition of the factorial function. (b) The denotational meaning of this function. $\perp$ denotes non-termination.

be mapped to the function f in Figure 3.1b. The idea is to abstract away details of a program, such as the names of variables or the algorithm and concentrate purely on the effects of the fragment.

A denotational way of looking at a function definition is as a fixed-point equation. For example, the recursive definition of the `fact` function in Figure 3.1a can be thought of as an equation,

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot f(x-1) & \text{otherwise,} \end{cases} \tag{3.1}$$

and the meaning of the recursive function definition is a function f that satisfies (3.1). More abstractly, a recursive function definition is a function F that transforms a function to a function. In this way, (3.1) can be written more simply as

$$f = \mathrm{F}(f). \tag{3.2}$$

An obvious question to ask is whether (3.2) has a solution and, if so, is it unique? It turns out that when f is a member of a complete partial order, a particular form of ordered set discussed in Section 3.2.1, and the function F is continuous, a concept discussed in Section 3.2.2, (3.2) always has a unique least solution, making this a reasonable way to interpret a function definition.

Zero-delay feedback looks like recursion, so I use this approach to handling recursive definitions to define SR systems with feedback. In my case, f is a vector-valued function defining the values on the communication channels, and F corresponds to evaluating all of the blocks in parallel. I present the details in Section 3.3.

Kahn's formal semantics for his concurrent dataflow language [38] (see Section 2.4.1) also use the fixed-point approach. He interprets each process as a function defined on potentially infinite streams of data, which represent the contents of the FIFOs his processes use to communicate. The semantics of a system with feedback is the solution to a fixed point equation defined on these streams.

Kahn's fixed point considers the whole execution history of the system. The values on the streams form a complete partial order—a set whose elements have a notion of "definedness"—under a prefix ordering. E.g., the stream 01 is less defined than the stream 01101. Under this prefix ordering, any process that waits when reading from an empty FIFO computes a continuous function, meaning that when more input is presented to a process, it may not change or reduce the amount of data it has already produced, nor may it wait forever. By requiring all processes to use such blocking reads, Kahn's systems are provably deterministic.

There is much more to denotational semantics. See the books by Winskel [73], Gunter [29], Stoy [65], Schmidt [60], and Allison [1].

3.1.2 Circuit Simulation

Circuit simulation has traditionally proceeded along two paths. Analog simulation attempts to model virtually all circuits, whereas digital or switch-level simulation treats only a restricted class of circuits

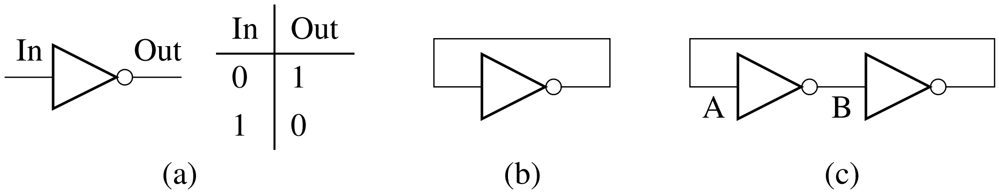


Figure 3.2 (a) An ideal binary inverter and its truth table. (b) A feedback circuit with no stable states. (c) A feedback circuit with two stable states.

in exchange for simulation speed. Progress along each path has often come from borrowing ideas from the other. Simpler models and the use of digital approximations has expedited analog simulation; more realistic models and other analog simulation techniques have improved digital simulation accuracy.

Analog simulation techniques are based on models from applied physics and use continuous mathematics. Pederson [55] provides an excellent historical review. A circuit is typically modeled as a system of ordinary non-linear differential equations and solved by a numerical integration method such the Trapezoidal Rule or a Runge-Kutta method. Newton's method is used on the resulting nonlinear systems, and LU decomposition or a sparse linear system solver is applied to the resulting linear systems.

By contrast, digital circuit simulators work with circuits that look very much like my SR systems: ideal (zero-delay) gates with an associated discrete function and well-defined inputs and outputs. Simulation consists of evaluating each gate (computing its outputs as a function of its inputs) in a topological order starting at the inputs to the circuit. Circuits with feedback, however, present a problem because their gates have no topological order.

Feedback in ideal zero-delay digital circuits present some of the same problems as it does in any zero-delay environment, including SR (see Section 1.3.1). The two major problems, contradiction and ambiguity, are shown in Figure 3.2. What value should the wire in Figure 3.2b take? The inverter makes its output the opposite of its input, but the wire forces all its connections to the same value. A physical realization of this circuit might find a stable intermediate voltage that is neither clearly 0 nor 1, or it might oscillate, but neither behavior fits into the ideal world of 0s and 1s. By contrast, there are two obvious possible states for the feedback circuit in Figure 3.2c. Node A could be either 0 or 1, and the circuit would be stable provided Node B is the opposite. Such behavior can be useful—this is how state-holding elements, such as static RAM cells, are built. However, this also deviates from the zero-delay digital model since the behavior of such circuits is a function of time as well as their inputs.

These problems have been addressed by making the digital circuit model more closely approximate the analog circuit model of a system of ordinary differential equations that must be solved rather than simply evaluated. Bryant's switch-level model [13] typifies this approach. He treats MOS transistors as bidirectional switches and treats a circuit as a network of nodes and switches. Each node has a weight to model its capacitance; switches have strengths to model their on-state conductance. To model a situation such as Figure 3.2b, he introduces a third node value, X, that "represents an uncertain or invalid node logic level or transistor conductance" and works in almost the same way as the undefined values in SR. He formulates the circuit as a sparse system of linear equations on a restricted, discrete domain and solves the resulting fixed-point equation using a relaxation method. His later work [11, 12] showed how this model can be cast purely as binary equations that can be solved even more rapidly on digital computers, producing the efficient switch-level simulator COSMOS [14].

Such three-valued logic has long been used in the study of asynchronous circuits. Here, the primary challenge is analyzing race conditions, where the behavior of a circuit is governed by unpredictable gate delays. Brzozowski and Seger [15] present a comprehensive

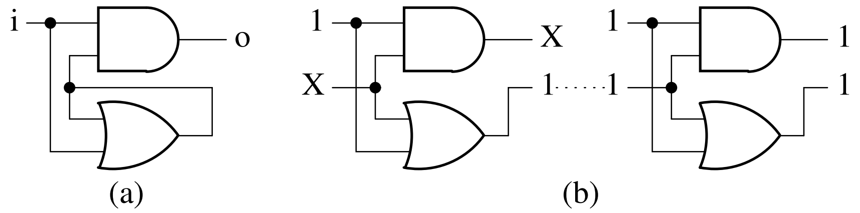


Figure 3.3 (a) A simple cyclic combinational circuit in which o follows i. (b) Malik's procedure for simulating the circuit, showing the output is 1 with when the input is 1.

theory. They start with a simple, conservative model of circuits that consist of ideal zero-delay binary gates and delay elements. These elements use an up-bounded inertial model in which the maximum delay is bounded, the minimum delay is not, and incoming transients shorter than the delay may be ignored. Next, they show that a model called General Multiple-Winner (GMW), which models the circuit as a nondeterministic* finite-state machine whose states represent the output of all delay elements, accurately captures the behavior of a circuit represented with the more detailed model. Finally, they show [63] that ternary simulation, based on Eichelberger's algorithm [26], accurately captures the behavior of a GMW circuit. In his algorithm, a third value models a signal that could be 0 or 1, effectively representing sets of states in the GMW model. This third value is essentially the same as the "undefined" value in SR.

Malik [46, 47] gives an algorithm based on ternary (X-valued) simulation that shows when a combinational circuit (i.e., one without explicit state-holding elements) with feedback is stateless. For example, the circuit in Figure 3.3a is combinational and cyclic. He applies ternary simulation after breaking all feedback arcs. A stable state of the circuit is found by first applying X's to the feedback arc inputs, simulating the (acyclic) circuit, and feeding the feedback outputs to

*The model uses nondeterminism to capture how unpredictable delays may produce varying behavior.

the inputs before simulating again. This process continues until the values on the feedback arcs are stable. Figure 3.3b illustrates this procedure. Malik shows that the monotonic nature of the gates ensures that this process will converge, a result almost identical to the one I present in Chapter 4.

Shiple, Berry, and Touati [64] extend Malik's work to sequential circuits—those with latches. They observe that although a cyclic circuit may not be well-behaved for certain inputs, if these inputs never appear the circuit should be considered well-behaved. To determine which inputs could appear, they use a Binary Decision Diagram-based state-space exploration scheme. Initially, they assume only the reset state is reachable and check if any of the possible inputs leads to a circuit in which an X remains (i.e., is not well-behaved). Then they repeat the procedure, adding all newly-discovered states to the reachable state list. The procedure terminates when either a poorly-behaved configuration is reached, or when the set of reachable states remains unchanged.

Berry first observed the connection between cyclic combinational circuits and causality problems in his Esterel programming language. The latest Esterel compiler (v5) uses this technique to test whether a program contains a paradox. Esterel is discussed in Section 2.3.5.

3.2 Mathematical Foundation

In this section, I present a series of well-known definitions and theorems that set the stage, mathematically, for Section 3.3, where I show SR systems are deterministic. Here, I present three main concepts. A *complete partial orders*, or CPO, is a set with an abstract notion of the amount of "information" in each element. Applying a *monotonic function* to an element of such a set always increases the amount of information unless it is a *fixed point*, in which case the element is unchanged.

Davey and Priestley's textbook on order in mathematics [23] is perhaps the best general introduction to this subject. The programming language semantics community is the other main source. I recommend the books by Winskel [73] (my primary source of notation) and Gunter [29]. Others include Allison [1], the very readable unpublished book by Turbak, Gifford, and Reistad [70], and chapters in the second volume of the *Handbook of Theoretical Computer Science* [71].

3.2.1 Complete Partial Orders

Definition 1 *A **partially-ordered set** or **poset** is a set S with a **partial order relation** $\sqsubseteq$ that satisfies*

- $x \sqsubseteq x$ *(Reflexive)*
- $x \sqsubseteq y$ *and* $y \sqsubseteq x$ *implies* $x = y$ *(Antisymmetric)*
- $x \sqsubseteq y$ *and* $y \sqsubseteq z$ *implies* $x \sqsubseteq z$ *(Transitive)*

The partial order relation $\sqsubseteq$ can be pronounced "approximates" or "is weaker than." It imposes some order on the members of S, but is less restrictive than a total order such as $\leq$. In particular, it is possible for two members of S to be incomparable, i.e., for neither $x \sqsubseteq y$ nor $y \sqsubseteq x$ to hold.

Virtually everything in this remainder of this chapter is a member of some partially-ordered set. The values in SR communication channels and even the block functions are members of posets.

A poset can be depicted with a Hasse diagram, such as that in Figure 3.4. An upward line is drawn between a pair of members x and y when $x \sqsubseteq y$, but lines implied by the transitive or reflexive rules are not drawn to simplify the diagram.

Definition 2 *An **upper bound** of a set T is an element u such that $t \sqsubseteq u$ for all $t \in T$. A **least upper bound** of a T, denoted $\sqcup T$, is an element l such that $l \sqsubseteq u$ for all upper bounds u.*

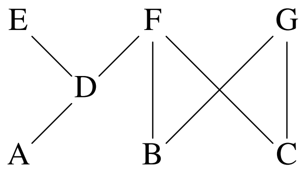


Figure 3.4 A Hasse diagram for a partially-ordered set. Here, $A \sqsubseteq D$, $D \sqsubseteq E$, $D \sqsubseteq F$, $B \sqsubseteq F$, $B \sqsubseteq G$, $C \sqsubseteq F$, $C \sqsubseteq G$. By the transitive rule, $A \sqsubseteq E$, but no line is drawn. A and B are incomparable.

The least upper bound of a set can be thought of as its limit. It will help define continuous functions, which are limit-preserving. Also, a later theorem will show that the solution to a fixed point equation is the least upper bound of a certain set.

Proposition 1 *The least upper bound of a set, if it exists, is unique.*

Proof Let u and v be two least upper bounds of a set S. For this to be true, $u \sqsubseteq v$, since u is a least upper bound, and similarly $v \sqsubseteq u$. Since $\sqsubseteq$ is antisymmetric, we must have $u = v$. ■

In Figure 3.4, the set $\{A, D\}$ has upper bounds D, E, and F, and a least upper bound D. The set $\{B, C\}$ has upper bounds F and G, but no least upper bound. The set $\{F, G\}$ has no upper bounds since there is no element above both.

Definition 3 *A* ***chain*** *is a totally-ordered set C, i.e., for all $x, y \in C$, either $x \sqsubseteq y$ or $y \sqsubseteq x$.*

Intuitively, a chain is a sequence of elements that are growing more defined. It appears as an upward path in a Hasse diagram. In Figure 3.4, $\{A, E\}$ and $\{A, D, F\}$ are chains. Although the members of $\{A, F, C\}$ are along a path, it is not a chain because neither $A \sqsubseteq C$ nor $C \sqsubseteq A$.

Definition 4 *A poset in which every chain in S has a least upper bound in S is a* ***complete partially-ordered set*** *or* ***CPO****.*

Complete partial orders are a class of well-behaved posets where ascending sequences always have limits. The posets in this chapter are all CPOs.

Proposition 2 *The least upper bound of a finite chain always exists and is its unique largest element.*

Proof The elements of a finite chain can be written $c_1 \sqsubseteq c_2 \sqsubseteq \cdots \sqsubseteq c_n$. Clearly, $c \sqsubseteq c_n$ for all c in the chain. Moreover, since $\sqsubseteq$ is antisymmetric, if there were a smaller c, it would satisfy $c \sqsubseteq c_n$ and $c_n \sqsubseteq c$, so $c = c_n$. Thus, c_n is the unique least upper bound. ∎

Corollary 1 *A poset with only finite chains is a CPO.*

I use this corollary frequently to ensure my posets are CPOs. This finite-chain restriction will also be instrumental in efficiently evaluating systems, a point I defer to Chapter 4.

Definition 5 *A* ***bottom element*** *of a poset, denoted $\perp$, is a member of S such that $\perp \sqsubseteq s$ for all $s \in S$. A poset with bottom is* ***pointed****.*

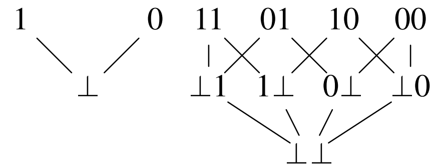
The bottom element of a poset is its least-defined member, representing “undefined.” Chains often start at $\perp$.

Although the bottom element of each poset is different, I will use the single symbol $\perp$ to represent them all. The meaning should be clear from context.

Proposition 3 *A bottom element, if it exists, is unique.*

Proof Assume b_1 and b_2 are bottom elements. By definition $b_1 \in S$ and $b_2 \in S$, and since b_1 is a bottom element, $b_1 \sqsubseteq b_2$. Similarly, $b_2 \sqsubseteq b_1$. Since $\sqsubseteq$ is antisymmetric, it follows that $b_1 = b_2$. ∎

The following proposition shows how vector-valued CPOs can be constructed from scalar-valued ones. Figure 3.5 shows an example.



(a) (b)

Figure 3.5 Using Proposition 4 to build a vector-valued CPO. (a) A scalar CPO (b) Its vector-valued extension.

Proposition 4 *If D_1 and D_2 are CPOs, then $D_1 \times D_2$ is a CPO under the ordering*

$$(x_1, x_2) \sqsubseteq (y_1, y_2) \text{ iff } x_1 \sqsubseteq y_1 \text{ and } x_2 \sqsubseteq y_2 \tag{3.3}$$

and if $x^1 = (x_1^1, x_2^1)$, $x^2 = (x_1^2, x_2^2)$, ...,

$$\sqcup\{x^1, x^2, \ldots\} = (\sqcup\{x_1^1, x_1^2, \ldots\}, \sqcup\{x_2^1, x_2^2, \ldots\}).$$

Proof $D_1 \times D_2$ is a poset, since the order relation is

- Reflexive: Since $x_1 \sqsubseteq x_1$ and $x_2 \sqsubseteq x_2$, $(x_1, x_2) \sqsubseteq (x_1, x_2)$.
- Antisymmetric: If $(x_1, x_2) \sqsubseteq (y_1, y_2)$ and $(y_1, y_2) \sqsubseteq (x_1, x_2)$, it follows that $x_1 = y_1$ and $x_2 = y_2$, so $(x_1, x_2) = (y_1, y_2)$.
- Transitive: If $(x_1, x_2) \sqsubseteq (y_1, y_2) \sqsubseteq (z_1, z_2)$, then $x_1 \sqsubseteq z_1$ and $x_2 \sqsubseteq z_2$ and $(x_1, x_2) \sqsubseteq (z_1, z_2)$.

It is a CPO since a least upper bound exists for all chains. Let $(x_1^1, x_2^1) \sqsubseteq (x_1^2, x_2^2) \sqsubseteq \cdots$ be a chain in $D_1 \times D_2$. It follows that $x_1^1 \sqsubseteq x_1^2 \sqsubseteq \cdots$ and $x_2^1 \sqsubseteq x_2^2 \sqsubseteq \cdots$ are chains in D_1 and D_2 and that $X = (x_1, x_2) = (\sqcup\{x_1^1, x_1^2, \ldots\}, \sqcup\{x_2^1, x_2^2, \ldots\})$ exists. By definition, for all i $(x_1^i, x_2^i) \sqsubseteq X$, so X is an upper bound. Assume $Y = (y_1, y_2)$ is some other upper bound, so $x_1^i \sqsubseteq y_1$ and $x_2^i \sqsubseteq y_2$. By the definition of X, $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$, so $X \sqsubseteq Y$. Thus, X is the least upper bound. ■

The following proposition presents a way to build a poset of functions. Basically, if $f(x) \sqsubseteq g(x)$ for all possible values of x, then $f \sqsubseteq g$.

It is more difficult to build a CPO of functions, as Theorem 1 will illustrate.

Proposition 5 *Let D and E be posets, and let $f : D \to E$ and $g : D \to E$ be two functions. If*

$$f \sqsubseteq g \text{ iff } \forall x \in D \ . \ f(x) \sqsubseteq g(x), \tag{3.4}$$

then $\sqsubseteq$ (defined on the members of $D \to E$) is a partial order relation.

Proof The relation $\sqsubseteq$ is

- Reflexive: Since $f(x) \sqsubseteq f(x)$ for all $f(x)$, it follows that $f \sqsubseteq f$.
- Antisymmetric: If $f \sqsubseteq g$ and $g \sqsubseteq f$, $f(x) \sqsubseteq g(x)$ and $g(x) \sqsubseteq f(x)$ for all x. It follows that $f(x) = g(x)$ for all x, implying $f = g$.
- Transitive: If $f \sqsubseteq g$ and $g \sqsubseteq h$, then $f(x) \sqsubseteq g(x) \sqsubseteq h(x)$, so $f(x) \sqsubseteq h(x)$, implying $f \sqsubseteq h$.

so it is a partial order relation. ■

3.2.2 Monotonic and Continuous Functions

In this section, I introduce order- and limit-preserving functions on posets. Equations with such well-behaved functions usually have solutions, a point I discuss in Section 3.2.3.

Definition 6 *A function $f : D \to E$ between posets D and E is* ***monotonic*** *iff for all $x, y \in D$ such that $x \sqsubseteq y$, $f(x) \sqsubseteq f(y)$.*

A monotonic function is order-preserving. If presented with more information, it responds with additional, non-contradictory information. Note that if D has no comparable members, any f is trivially monotonic.

I introduce a shorthand for applying a function to every member of a set: $\{f(C)\} \equiv \{f(c) \mid c \in C\}$.

Definition 7 *A function $f : D \to E$ between CPOs D and E is* ***continuous*** *if for all all chains $C \subseteq D$, $f(\sqcup C) = \sqcup\{\, f(c) \mid c \in C\}$, or equivalently $f(\sqcup C) = \sqcup\{f(C)\}$.*

Since D is complete, we know $\sqcup C$ exists. This definition is saying, as a side effect, that $\sqcup\{f(C)\}$ also exists for a continuous function.

A continuous function is limit-preserving. The limit of a continuous function evaluated on a chain is equal to the function evaluated at the limit of the chain. The following proposition shows that continuous functions are a strict subset of the monotonic functions.

Proposition 6 *A continuous function is monotonic.*

Proof Let $f : D \to E$ be continuous, and let $x \sqsubseteq y \in D$. Since $\sqcup\{x, y\} = y$, and since f is continuous, $f(x) \sqsubseteq \sqcup\{f(x), f(y)\} = f(\sqcup\{x, y\}) = f(y)$. So $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ and f is monotonic. ■

The following proposition provides a way to ensure a monotonic function is continuous. This is useful because monotonicity is more intuitive and easier to check.

Proposition 7 *A monotonic function on a CPO with only finite chains is continuous.*

Proof Let $f : D \to E$ be a monotonic function, and let $C = \{c_1, \ldots, c_n\} \subseteq D$ be a chain where $c_1 \sqsubseteq c_2 \sqsubseteq \cdots \sqsubseteq c_n$. Because f is monotonic, we have $f(c_1) \sqsubseteq f(c_2) \sqsubseteq \cdots \sqsubseteq f(c_n)$. From Proposition 2, $\sqcup C = f(c_n)$, and so $f(\sqcup C) = f(c_n) = \sqcup\{f(C)\}$. It follows that f is continuous. ■

The following two propositions show continuity and monotonicity are closed under composition.

Proposition 8 *The composition $g \circ f$ of two continuous functions $f : D \to E$ and $g : E \to F$ is continuous.*

Proof Let C be a chain in D. Since f is continuous, $f(\sqcup C) = \sqcup\{f(C)\}$. Moreover, since C is a chain and f is continuous, $\{f(C)\}$ is also a chain. Since g is continuous, $g(\sqcup\{f(C)\}) = \sqcup\{g(f(C))\}$, and also $g(f(\sqcup C)) = \sqcup\{g(f(C))\}$. Hence $g \circ f$ is also continuous. ∎

Proposition 9 *The composition $g \circ f$ of two monotonic functions $f : D \to E$ and $g : E \to F$ is monotonic.*

Proof Since f is monotonic, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. Since g is monotonic, it follows that $g(f(x)) \sqsubseteq g(f(y))$. ∎

The following proposition provides a way to build a vector-valued continuous function from two continuous functions. Compare with Proposition 4, which builds vector-valued CPOs in a similar way.

Proposition 10 *Let D, E, and F be CPOs. If $f : D \to E$ and $g : D \to F$ are continuous, then $f \times g$ is continuous*

Proof Since E and F are CPOs, then $E \times F$ is a CPO by Proposition 4.

Let $x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ be a chain in D. Because f and g are continuous, $f \times g$ is continuous because

$$\begin{array}{l}
(f \times g)(\sqcup\{x_1, x_2, \ldots\}) \\
\quad = \quad (f(\sqcup\{x_1, x_2, \ldots\}), g(\sqcup\{x_1, x_2, \ldots\})) \\
\quad = \quad (\sqcup\{f(x_1), f(x_2), \ldots\}, \sqcup\{g(x_1), g(x_2), \ldots\}) \\
\quad = \quad \sqcup\{(f(x_1), g(x_1)), (f(x_2), g(x_2)), \ldots\} \\
\quad = \quad \sqcup\{(f \times g)(x_1), (f \times g)(x_2), \ldots\}.
\end{array}$$

∎

The following proposition provides a way to take the least upper bound of a chain of functions, which will be useful for working with higher-order functions (those functions whose domain and range are themselves functions).

Proposition 11 *Let D and E be CPOs, let $f_k : D \to E$ be continuous, and let $f_1 \sqsubseteq f_2 \sqsubseteq \cdots$. Then $g = \sqcup\{f_1, f_2, \ldots\}$ exists and $g(x) = \sqcup\{f_1(x), f_2(x), \ldots\}$.*

Proof From (3.4), $f_1(x) \sqsubseteq f_2(x) \sqsubseteq \cdots$ is a chain. Since E is a CPO, $g(x) = \sqcup\{f_1(x), f_2(x), \ldots\}$ exists. This is an upper bound since $f_k(x) \sqsubseteq g(x)$ for all k. Moreover, it is a least upper bound since if there was another upper bound h, then $f_k(x) \sqsubseteq h(x)$ for all k, but by definition of g, $g(x) \sqsubseteq h(x)$. ∎

The next fundamental theorem is necessary when working with higher-order functions. All of the useful results require working with elements of a CPO, and this provides a CPO of functions.

Theorem 1 *Let D and E be pointed CPOs. The set of all continuous total functions mapping D to E forms a pointed CPO.*

Proof Let **F** denote the set of continuous total functions from D to E. To show **F** is a pointed CPO, I will show it is a pointed poset and show that each chain of functions in **F** has a least upper bound that is a function in **F**, i.e., is a continuous function from D to E.

F is a pointed poset. Proposition 5 implies (3.4) is a partial order relation for **F**, and its bottom element is $\bot(x) = \bot$. This is a member of **F** since it is continuous: Let $C \in D$ be a chain. $\bot(\sqcup C) = \bot$, and $\sqcup\{\,\bot(c) \mid c \in C\} = \bot$. Moreover, it is the bottom element, since if there existed another function $b \sqsubseteq \bot$, then $b(x) \sqsubseteq \bot(x)$ for all x. However, by definition of $\bot$, it follows that $b(x) = \bot$.

An ascending chain $f_1 \sqsubseteq f_2 \sqsubseteq \cdots$ in **F** has a least upper bound g from Proposition 11.

The least upper bound g of an ascending chain in **F** is a monotonic function. If $x \sqsubseteq y$, it follows that $f_i(x) \sqsubseteq f_i(y) \sqsubseteq g(y)$ for all i. So $g(y)$ is an upper bound of $\{f_i(x)\}$, but since $g(x)$ is the least upper bound of $\{f_i(x)\}$, $g(x) \sqsubseteq g(y)$.

The least upper bound of an ascending chain in **F** is a continuous function. Let $x_1 \sqsubseteq x_2 \sqsubseteq$ be an ascending chain in D. Since D is a

CPO, $\sqcup\{x_1, x_2, \ldots\}$ exists. Moreover, since g is monotonic, $g(x_1) \sqsubseteq g(x_2) \sqsubseteq$ is an ascending chain in E, so $\sqcup\{g(x_1), g(x_2), \ldots\}$ also exists. To see they are equal, I will use the notation

$$\sqcup_i\{f_i(x)\} = \sqcup\{f_1(x), f_2(x), \ldots\}.$$

From the definition of $\sqcup$,

$$\begin{array}{rcl} \forall i, j \,.\, f_i(x_j) & \sqsubseteq & \sqcup_k\{f_k(x_j)\} \\ \forall i \,.\, \sqcup_l\{f_i(x_l)\} & \sqsubseteq & \sqcup_l\{\sqcup_k\{f_k(x_l)\}\} \\ \sqcup_k\{\sqcup_l\{f_k(x_l)\}\} & \sqsubseteq & \sqcup_l\{\sqcup_k\{f_k(x_l)\}\} \end{array}$$

and

$$\begin{array}{rcl} \forall i, j \,.\, f_i(x_j) & \sqsubseteq & \sqcup_l\{f_i(x_l)\} \\ \forall j \,.\, \sqcup_k\{f_k(x_j)\} & \sqsubseteq & \sqcup_k\{\sqcup_l\{f_k(x_l)\}\} \\ \sqcup_l\{\sqcup_k\{f_k(x_l)\}\} & \sqsubseteq & \sqcup_k\{\sqcup_l\{f_k(x_l)\}\} \end{array}$$

so

$$\sqcup_k\{\sqcup_l\{f_k(x_l)\}\} = \sqcup_l\{\sqcup_k\{f_k(x_l)\}\}.$$

Since the f_i are continuous, this implies

$$\begin{array}{rcl} \sqcup_k\{f_k(\sqcup_l\{x_l\})\} & = & \sqcup_l\{\sqcup_k\{f_k(x_l)\}\} \\ g(\sqcup_l\{x_l\}) & = & \sqcup_l\{g(x_l)\}, \end{array}$$

which shows g is continuous. ■

3.2.3 Least Fixed Points

Definition 8 *Let D be a poset, let $f : D \to D$ be a function, and let $x \in D$.*

- *If $f(x) \sqsubseteq x$, then x is a* ***prefixed point****.*
- *If $f(x) = x$, then x is also a* ***fixed point***
- *If x is a prefixed point and $x \sqsubseteq p$ for all prefixed points p, then x is a* ***least prefixed point****.*

• *If x is a fixed point and $x \sqsubseteq y$ for all fixed points y, then x is a* ***least fixed point****.*

The following well-known theorem* is key to everything in this dissertation. It will be used to show that an SR system always has a unique behavior, and its proof contains the fundamental idea used to evaluate the systems.

Theorem 2 *Let $f : D \to D$ be a continuous function on a pointed CPO D. Then*

$$\mathrm{fix}(f) \equiv \sqcup\{\bot, f(\bot), f(f(\bot)), \ldots, f^k(\bot), \ldots\} \tag{3.5}$$

exists and is both the unique least fixed point and the unique least prefixed point of f.

Proof $\{\bot, f(\bot), f^2(\bot), \ldots\}$ is a chain since by definition $\bot \sqsubseteq f(\bot)$ and

$$\begin{array}{rcl} f(\bot) & \sqsubseteq & f(f(\bot)) \\ f(f(\bot)) & \sqsubseteq & f^3(\bot) \\ & \vdots & \end{array}$$

because f is monotonic by Proposition 6. Since D is complete, the least upper bound of this chain, $\mathrm{fix}(f)$, exists. Furthermore, because f is continuous,

$$\begin{array}{rcl} f(\mathrm{fix}(f)) & = & f(\sqcup\{\bot, f(\bot), f^2(\bot), \ldots\}) \\ & = & \sqcup\{f(\bot), f(f(\bot)), f(f^2(\bot)), \ldots\} \\ & = & \sqcup\{\bot, f(\bot), f^2(\bot), \ldots\} \\ & = & \mathrm{fix}(f) \end{array}$$

so $\mathrm{fix}(f)$ is a fixed point and therefore also a prefixed point.

*It is similar to the Knaster-Tarski fixed point theorem, but that result only applies to functions defined on complete lattices—posets whose subsets always have both greatest lower and least upper bounds (See Davey and Priestley [23]). My CPOs are less structured than this.

Let x be another prefixed point, i.e., $f(x) \sqsubseteq x$. Since f is monotonic,

$$\begin{aligned}
\perp &\sqsubseteq x \\
f(\perp) &\sqsubseteq f(x) \quad \sqsubseteq x \\
f^2(\perp) &\sqsubseteq f^2(x) \sqsubseteq f(x) \sqsubseteq x \\
&\vdots \\
f^k(\perp) &\sqsubseteq x \\
&\vdots
\end{aligned}$$

so x is an upper bound of the chain $\{\perp, f(\perp), f^2(\perp), \ldots\}$. However, since $\mathrm{fix}(f)$ is the least upper bound of this chain, $\mathrm{fix}(f) \sqsubseteq x$. $\mathrm{fix}(f)$ is thus the least prefixed point, and since it is a fixed point, it is also the least fixed point.

■

3.3 The Semantics of SR Systems

The fundamental piece of computation in an SR system is a block—a vector-valued function with a fixed number of inputs and outputs. The meaning of such a block is obvious, but the meaning of a system (a composition of blocks) is less so. The primary result of this section is a procedure that transforms a system into a block. It forms a vector composed of inputs to the system and the outputs of every block, connects all the blocks to this vector, and finds the least function that produces consistent values for the outputs. I show this function is unique and that it behaves like a block.

Definition 9 *Let* $\mathbf{I} = I_1 \times \cdots \times I_n$ *and* $\mathbf{O} = O_1 \times \cdots \times O_m$ *be vectors of pointed CPOs. An n-input, m-output* ***block*** *b is a continuous vector-valued function from* $\mathbf{I}$ *to* $\mathbf{O}$.

Definition 10 *Let* $b : \mathbf{I} \to \mathbf{O}$ *be a block, let* $\mathbf{J} = J_1 \times \cdots \times J_a$ *be a vector of pointed CPOs, and let* $w_1, \ldots, w_n$ *be a sequence such that* $w_k \in \{1, \ldots, a\}$. *If*

*$J_{w_k} \subseteq I_k$, then the block b **connected** to $\mathbf{J}$ with connections $w_1, \ldots, w_n$ is the function $c : \mathbf{J} \to \mathbf{O}$ such that*

$$c(j_1, \ldots, j_a) = b(j_{w_1}, \ldots, j_{w_n}),$$

where $(j_1, \ldots, j_a) \in \mathbf{J}$.

The following definitions are illustrated in Figure 3.6.

Definition 11 *Let $b_1 : \mathbf{I}_1 \to \mathbf{O}_1$, $b_2 : \mathbf{I}_2 \to \mathbf{O}_2$, ..., $b_s : \mathbf{I}_s \to \mathbf{O}_s$ be a collection of blocks, let $\mathbf{I} = I_1 \times \cdots \times I_n$ be a vector of pointed CPOs, let $\mathbf{O} = \mathbf{O}_1 \times \cdots \times \mathbf{O}_s$, and let $\mathbf{J} = \mathbf{I} \times \mathbf{O}$. The **open system** d of these blocks is the function $d : \mathbf{J} \to \mathbf{O}$ such that*

$$d(j) = c_1(j) \times c_2(j) \times \cdots \times c_s(j).$$

where c_k is the block b_k connected to the vector $\mathbf{J}$, and $j \in \mathbf{J}$.

Definition 12 *The **SR System** of an open system d is the least function $e : \mathbf{I} \to \mathbf{O}$ that satisfies*

$$e(i) = d(i, e(i)). \tag{3.6}$$

The remainder of this section is devoted to showing that SR systems are deterministic by showing that (3.6) always has a unique solution. This is a fixed-point equation with a *function e* as the argument, i.e.,

$$e = \mathcal{B}(e), \tag{3.7}$$

where $\mathcal{B} : (\mathbf{I} \to \mathbf{O}) \to (\mathbf{I} \to \mathbf{O})$ is a function that transforms a function to a function. If $f : \mathbf{I} \to \mathbf{O}$ is a function, then $\mathcal{B}(f) : \mathbf{I} \to \mathbf{O}$ is the function

$$\mathcal{B}(f)(i) = d(i, f(i)). \tag{3.8}$$

I will use Theorem 2 to show $\mathcal{B}$ has a unique least fixed point. To do this, I will show its domain is a pointed CPO, i.e., block functions form a complete partial order, and the function $\mathcal{B}$ is continuous.

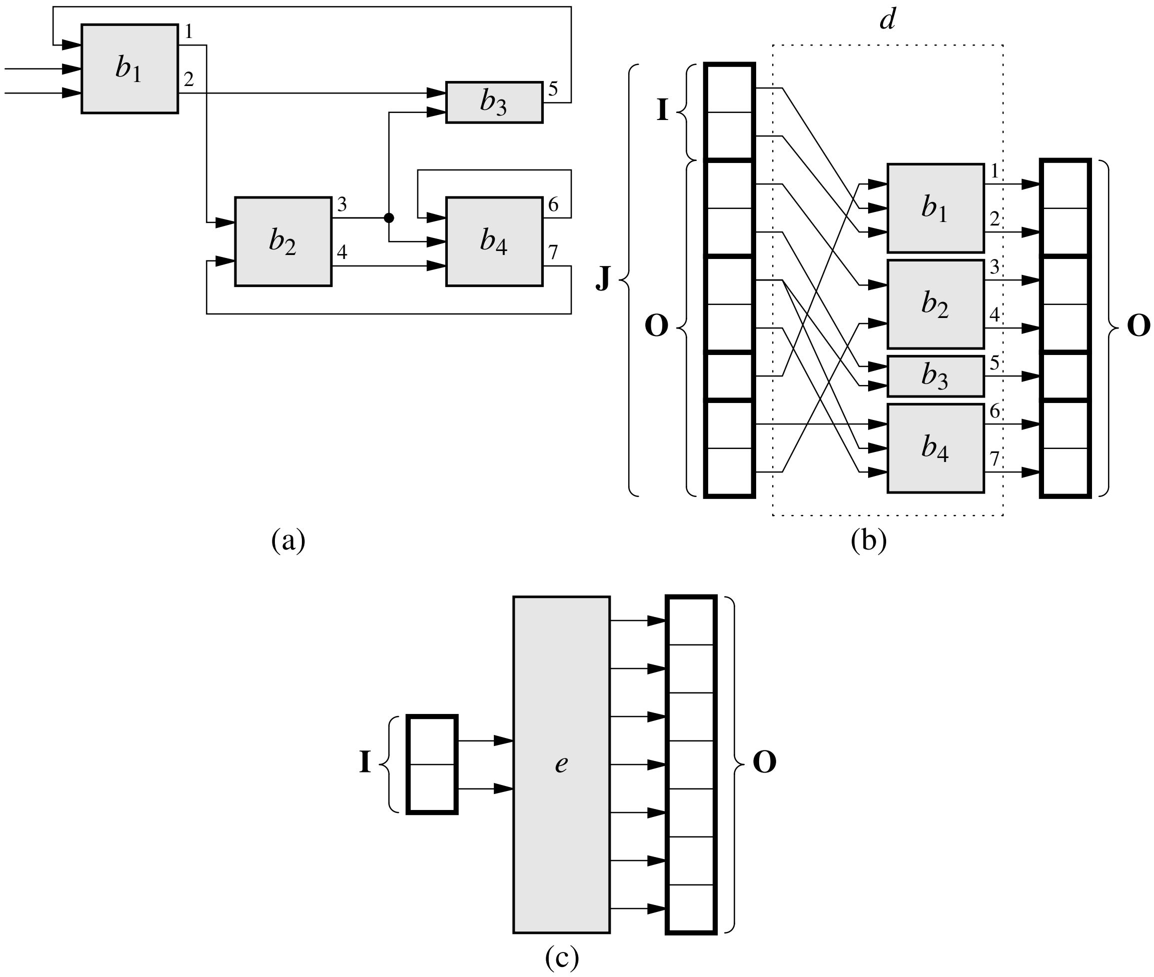


Figure 3.6 (a) An SR system. (b) The corresponding open system. (c) The corresponding block.

Lemma 1 *The set of all block functions forms a pointed CPO.*

Proof Let b be a block. By Proposition 4 and induction, both $\mathbf{I} = I_1 \times \cdots \times I_n$ and $\mathbf{O} = O_1 \times \cdots \times O_m$ are pointed CPOs. Since b is continuous, Theorem 1 shows these functions form a pointed CPO. ■

Lemma 2 *$\mathcal{B}(f)$ is a block function if f is.*

Proof From their definition, the input and output domains of $\mathcal{B}(f) : \mathbf{I} \to \mathbf{O}$ are vectors of pointed CPOs. Moreover, since f and $c_1, \ldots, c_s$ are continuous, and $\mathcal{B}(f)(i) = (c_1(i, f(i)), \ldots, c_s(i, f(i)))$, it follows from Propositions 8 and 10 that $\mathcal{B}(f)$ is continuous. ■

Lemma 3 *$\mathcal{B}$ is continuous.*

Proof First, note that d is continuous since $c_1, \ldots, c_s$ are continuous since $b_1, \ldots, b_s$ are.

Next, let $F = \{f_1, f_2, \ldots\}$ be a chain in the CPO of continuous functions $\mathbf{I} \to \mathbf{O}$. $\mathcal{B}$ is continuous because

$$
\begin{aligned}
\mathcal{B}(\sqcup F)(i) &= d(i, (\sqcup F)(i)) \\
&= d(i, (\sqcup_k \{f_k\})(i)) \\
&= d(i, \sqcup_k \{f_k(i)\}) \\
&= \sqcup_k \{d(i, f_k(i))\} \\
&= \sqcup_k \{\mathcal{B}(f_k)(i)\} \\
&= \sqcup \{\mathcal{B}(F)(i)\}
\end{aligned}
$$

by definition of $\mathcal{B}$, Proposition 11, because d is continuous, and by Proposition 4. ■

Theorem 3 *SR Systems are deterministic, i.e., (3.6) has a unique solution that is a block.*

Proof From Lemmas 1 and 2, it follows that the domain of blocks and the range of $\mathcal{B}(b)$ is a pointed CPO, and from Lemma 3, $\mathcal{B}$ is a continuous function on this CPO. From Theorem 2, it follows that (3.6) has a unique least solution that satisfies Definition 9. ■

Chapter 4

Execution

The fundamental qualities for good execution
of a plan is first; intelligence;
then discernment and judgment,
which enable one to recognize
the best method as to attain it;
the singleness of purpose;
and, lastly, what is most essential of all,
will—stubborn will.
—Ferdinand Foch

EXECUTING an SR system for an instant amounts to solving a fixed-point equation. This is challenging because it needs to be solved efficiently and predictably, and because of heterogeneity, it is only possible to evaluate the function whose least fixed point is being computed.

I solve this problem by computing a schedule for an SR system—a fixed execution sequence for its blocks that solves the fixed-point equation in accordance with the semantics in Chapter 3. Once scheduled, a system can be simulated by running the schedule, or synthesized using a block code generation technique where the code for each block is inlined according to the schedule.

First and foremost, a schedule must make the system behave according to the semantics in Chapter 3, but minimizing a schedule's running time is also important. An SR system must respond to inputs before more arrive, so a system will fail if it is too slow relative to its environment. Hence, only worst-case execution time is worth optimizing, and I do not consider any techniques that could expedite execution for certain combinations of states and inputs. Besides, the heterogeneity of the blocks generally precludes most of these techniques because little is known about their function.

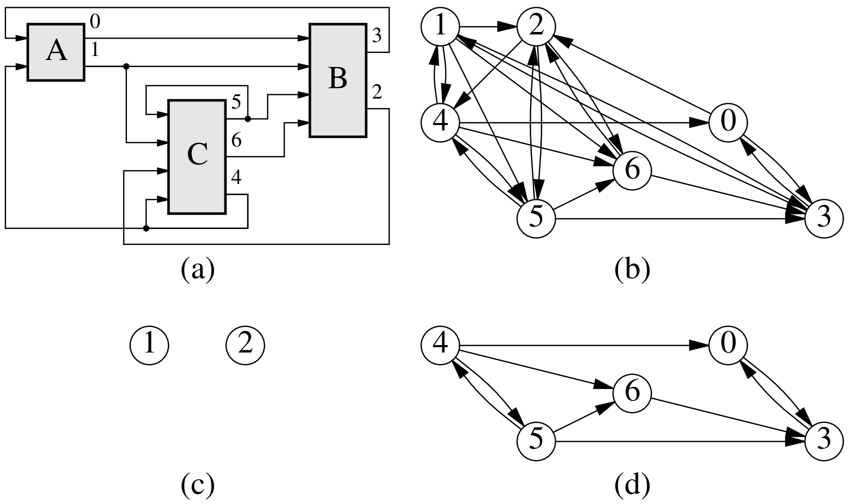
These schedules are computed by recording the sequence of blocks evaluated by an algorithm that computes the least fixed point. Since this algorithm does not make any decisions based on inputs or states, the schedule is correct for any input or state.

The algorithm uses a divide-and-conquer strategy to find the least fixed point using a minimum number of function evaluations. Splitting a function into pieces, finding the least fixed point of each piece, and combining them to form the result is usually more efficient than tackling the whole function.

The algorithm finds efficient schedules by carefully choosing the place to split the function. Although this choice can greatly affect a schedule's execution time, the algorithm produces a correct schedule for any choice of where to split, so efficiency can be optimized without affecting correctness.

Figure 4.1 shows the complete process of computing a schedule. A dependency graph representing the communication in the system is first derived. It is then decomposed into strongly connected components (a trivial step in this example, since the graph happens to be strongly connected), and a carefully chosen set of vertices (here, 1 and 2) is removed from each component and the process is repeated. The steps in the decomposition are recorded in a schedule, which is then modified slightly to speak of block, as opposed to block output, evaluations.

This chapter is divided into three parts. I review related work, including chaotic iteration and graph-based function evaluation, in the first. In the second, I present the divide-and-conquer least fixed point computation algorithm along with theorems that show it is correct.



$(\,1\;2\;.\;(\,4\;.\;5\,)^1\;6\;(\,0\;.\;3\,)^1\,)^2$ $\qquad$ $(A\;.\;(C\;.\;C)^1\;C\;(A\;.\;B)^1\,)^2$

(e) (f)

C C C C B A B A C C C C B A B A C C C C B A B

(g)

Figure 4.1 A complete example of scheduling an SR system. (a) The system. (b) Its (strongly connected) dependency graph. (c) The vertices removed by the scheduler to break strong connectivity. (d) The graph that remains. (e) The schedule. Superscripts denote the number of iterations. (f) The schedule after its transformation to block evaluations. (g) The sequence of blocks to evaluate each instant.

In the third, I consider the problem of finding efficient schedules: I characterize minimal-cost schedules, present an exact branch-and-bound algorithm for computing them, present a heuristic that prunes the number of branches considered, and present experimental results that show both exact and heuristic splitting algorithms are practical.

4.1 Related Work

My algorithm for computing least fixed points can be viewed as a chaotic iteration scheme. The proof of Theorem 2 from the last chapter suggests that the least fixed point can be found by repeatedly evaluating the function. Chaotic iteration is a variation that evaluates individual elements of a vector-valued function in some order. A schedule is then a chaotic iteration strategy—an order in which to evaluate the parts.

Chaotic iteration has long been used for solving systems of linear equations. The Gauss-Seidel method is a familiar example that solves the matrix equation $Ax = b$ by repeatedly evaluating

$$x_i := \frac{1}{a_{ii}} \left[b_i - \sum_{j=1,\ldots,n,j\neq i} a_{ij} x_j \right]$$

for $i = 1, \ldots, n, 1, \ldots, n, 1, \ldots$. Such an evaluation order is a chaotic iteration strategy, and generally does not affect correctness. As such, the technique works well on parallel computers with little synchronization. Chazan and Miranker [21] were two of its early pioneers. Unfortunately, most of the techniques developed for these continuous problems do not translate well to the discrete-valued domains in SR systems.

Robert's [58] approach to discrete iteration is very abstract and general. However, since he places so few restrictions on the functions, his results are not strong enough to be useful here. In particular, he can only predict convergence rates for systems that can be topologically ordered.

Chaotic iteration as an evaluation scheme resembles the delta delay model of the VHDL [45] and Verilog [69] discrete-event simulation languages. In these, each instant of simulated time is broken into a sequence of delta timesteps to simulate zero delay elements. The big difference is that in these languages, the behavior of the system can depend on the order of events in these delta timesteps. These are not always specified in the language, which can lead to nondeterminism. Even worse, it is impossible in general to predict how many delta timesteps are required in a particular instant, or even whether the number is bounded for all instants. The execution scheme presented here for SR systems suffers from none of these problems.

In my master's thesis [25], I had shown these techniques are applicable to the execution of synchronous languages. There, I presented a compiler for the Esterel language (see Section 2.3.5 on Page 21) loosely based on these techniques. It used a chaotic iteration scheme to find the fixed point of a monotonic function derived directly from the program source. It used a dynamic evaluation scheme, and its scheduler did not attempt to improve efficiency or predictability.

Many researchers have observed that self-referential systems can be evaluated more efficiently after being decomposed into strongly connected components. For example, Buhl et al. [18] applies this to nonlinear differential algebraic systems that arise in the simulation of heating and cooling systems in buildings. Jones [37] applies this to evaluating circular attribute grammars used for checking programs' static semantics. However, most of these techniques simply apply a brute-force evaluation technique to each strongly connected component.

Bourdoncle [9] proposes the Weak Topological Order (WTO), essentially a recursive strongly connected component decomposition. A WTO is a parenthesized linear ordering of the vertices in a directed graph such that back edges (i.e., those from later to earlier vertices) only land on the first vertex in each parenthesized component, called the *head* of the component. For example, a WTO for the graph in Figure 4.6 is

$$(\underline{7}\ (\underline{5}\ 2\ 1\ 3)\ 4\ 6).$$

Here, the head of each component is underlined. His recursive evaluation strategy evaluates each parenthesized component by evaluating all its contained vertices until its head has converged. Any inner components are brought to convergence for each evaluation of the outer component.

Bourdoncle's approach largely inspired mine, but the problem he solves is slightly different. He is not concerned with the predictability of the fixed-point evaluation scheme. Moreover, beyond a bound, he does not address the quality of a WTO, nor does he give an algorithm that finds a provably optimal WTO.

Since minimizing the execution time of an SR system is critical for ensuring the synchrony hypothesis holds, Bourdoncle's results are not sufficient. A fundamental limitation of his WTOs is that their heads are limited to single vertices, which is sub-optimal for certain graphs (e.g., Figure 4.15). My algorithms consider both single- and multiple-vertex heads.

4.2 Finding the Least Fixed Point

From Definition 12, executing an SR system requires evaluating a least fixed point in each instant. This section concentrates on an algorithm for doing this correctly, and concludes with a proof of its correctness. Ultimately, the behavior of this algorithm will be recorded to produce a schedule. The next section is devoted to the problem of making this algorithm find efficient schedules.

$o = e(i)$ in each instant, where i is the vector of inputs, o is the vector of outputs, and e is the least function that satisfies

$$e(i) = d(i, e(i)),$$

where d is the aggregation of block functions from Definition 10.

Since I am using the pointwise partial ordering on functions, (3.4), the least function is the one that takes the least value at each value of i. Thus, evaluating $e(i)$ for a particular i amounts to finding the least o such that

$$o = d(i, o).$$

For convenience, define $F(o) = d(i, o)$. The objective is now the least o such that

$$o = F(o).$$

By definition of fix, the solution to this is

$$o = \mathrm{fix}(F). \tag{4.1}$$

In light of Theorem 3, it is not surprising that (4.1) is well-defined. This follows from Theorem 2 since F is continuous (since d is continuous) and its domain, $\mathbf{O}$, is a pointed CPO that is the vector of all outputs.

4.2.1 Iterative Evaluation

In this section, I present a way of directly computing the fixed point of a function. This approach, described in Theorem 5, is an iterative approach applicable to any function, inspired by the proof of Theorem 2.

The height of a CPO, as defined below, provides a bound on the number of function evaluations necessary to find the least fixed point. The height is a natural measure since it is additive—the height of a vector-valued CPO is the sum of the heights of its components (Theorem 4 and Corollary 2 show this). Although many useful CPOs have infinite chains (see Davey and Priestly [23]), I do not consider them here, as the finite-height assumption is required to make evaluation predictable.

Definition 13 *The **height** of a pointed CPO D, written $h(D)$, is one less than the length of its longest chain. If $F : D \to D$ is a function, then $h(F) = h(D)$.*

Theorem 4 *If A and B are pointed CPOs, then $A \times B$ is a pointed CPO with height $h(A \times B) = h(A) + h(B)$ under the component-wise ordering (Equation (3.3)).*

Proof From Proposition 4, $A \times B$ is a CPO. Its bottom element is $(\bot, \bot)$, so it is also pointed.

Let $\bot \sqsubseteq a_1 \sqsubseteq \cdots \sqsubseteq a_{h(A)}$ be a chain in A, and let $\bot \sqsubseteq b_1 \sqsubseteq \cdots \sqsubseteq b_{h(B)}$ be a chain in B. It follows that

$$(\bot,\bot) \sqsubseteq \underbrace{(a_1,\bot) \sqsubseteq \cdots \sqsubseteq (a_{h(A)},\bot)}_{h(A)\text{ elements}} \sqsubseteq \underbrace{(a_{h(A)},b_1) \sqsubseteq \cdots \sqsubseteq (a_{h(A)},b_{h(B)})}_{h(B)\text{ elements}}$$

is a chain of length $h(A) + h(B) + 1$.

Now assume there exists a chain of length $k > h(A) + h(B) + 1$ in $A \times B$, i.e.,

$$(a_1, a_1) \sqsubseteq (a_2, b_2) \sqsubseteq \cdots \sqsubseteq (a_k, b_k).$$

From (3.3), it follows that $a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq a_k$ and $b_1 \sqsubseteq b_2 \sqsubseteq \cdots \sqsubseteq b_k$. However, since the height of A is $h(A)$ and the height of B is $h(B)$, these sequences can have at most $h(A) + 1$ and $h(B) + 1$ unique elements respectively. Moreover, since $(a_i, b_i) \neq (a_{i+1}, b_{i+1})$ for $i = 1, \ldots, k-1$, there can be at most $(h(A) - 1) + (h(B) - 1) + 1$ unique pairs in the ascending sequence—at least one of the elements must be different, and there are only $h(A)$ and $h(B)$ possible choices for each.

There exists a chain of height $h(A) + h(B) + 1$, and none may be longer, so the height of $A \times B$ is $h(A) + h(B)$. ■

Corollary 2 *Let $A_1 \times A_2 \times \cdots \times A_k$ be a pointed CPO of pointed CPOs A_1, A_2, …, A_k. Then*

$$h(A_1 \times A_2 \times \cdots \times A_k) = \sum_{i=1}^{k} h(A_i). \tag{4.2}$$

The following provides a general technique for iterating toward a fixed point. The iteration may start at any point c below the actual fixed point where the function is increasing and proceeds by evaluating $F(c)$, $F(F(c))$, …, $F^k(c)$, etc. The number of iterations is bounded by the height of the CPO on which F is defined.

Theorem 5 *Let D be a pointed CPO, and let $F : D \to D$ be a continuous function. If $c \sqsubseteq F(c)$ and $c \sqsubseteq \mathrm{fix}(F)$, then $F^{h(F)}(c) = \mathrm{fix}(F)$.*

Proof By Theorem 2, $\mathrm{fix}(F)$ exists. Since $c \sqsubseteq F(c)$ and F is monotonic, $C = \{c, F(c), F(F(c)), \ldots\}$ is a chain:

$$\begin{array}{rcl} c & \sqsubseteq & F(c) \\ F(c) & \sqsubseteq & F(F(c)) \\ & \vdots & \end{array}$$

Moreover, the least upper bound of this chain is a fixed point since F is continuous:

$$\begin{array}{rcl} F(\sqcup\{c, F(c), F^2(c), \ldots\}) & = & \sqcup\{F(c), F(F(c)), F(F^2(c)), \ldots\} \\ & = & \sqcup\{c, F(c), F^2(c), \ldots\} \end{array}$$

By definition, the least upper bound must approximate this, so $\mathrm{fix}(F) \sqsubseteq \sqcup C$. Moreover, since $c \sqsubseteq \mathrm{fix}(F)$, it must be that

$$\begin{array}{rcl} c & \sqsubseteq & \mathrm{fix}(F) \\ F(c) & \sqsubseteq & F(\mathrm{fix}(F)) = \mathrm{fix}(F) \\ F^2(c) & \sqsubseteq & \mathrm{fix}(F) \\ & \vdots & \end{array}$$

so $\sqcup C \sqsubseteq \mathrm{fix}(F)$. Thus, $\sqcup C = \mathrm{fix}(F)$.

Let the height of D be $h = h(F)$. By definition, C may contain at most $h+1$ distinct elements. There are two possibilities:

1. If $c \neq F(c) \neq \cdots \neq F^h(c)$, then C is a chain with $h+1$ elements and $\sqcup C = F^h(c) = \mathrm{fix}(F)$.

2. $F^{k-1}(c) = F^k(c)$ for some $k \leq h$. This means $F^{k-1}(c)$ is a fixed point, and since $k \leq h$, it follows that $F^h(c) = F^{k-1}(c)$.

■

4.2.2 Chaotic Iteration

In chaotic iteration, the least fixed point of a vector-valued function is found through repeated evaluation of parts of the function. This section contains some notation that will clarify what I mean by a "part" of a function, and concludes with a proof of properties I call the Chaotic Iteration Invariants, which shows that iterating to a least fixed point results in a monotonically-increasing sequence that never overshoots the least fixed point.

The basic operation in chaotic iteration is the evaluation of some dimensions of a vector-valued function. To facilitate the discussion of this, I introduce the following notation. Consider a domain D of n-dimensional vectors: $D = D_1 \times D_2 \times \cdots \times D_n$. These will represent the values on the communication channels. Frequently, I will be speaking of some group of channels, which I denote by a set $S = \{s_1, \ldots, s_k\}$, a subset of $\{1, \ldots, n\}$ such that $1 \leq s_1 < s_2 < \cdots < s_k \leq n$. I often refer to all the remaining channels, i.e., the complement ($\{1, \ldots, n\} - S$) of this set, which I write as $\bar{S}$. Juxtaposition denotes set intersection, i.e., $AB = A \cap B$.

Let $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_n)$, and $z = (z_1, \ldots, z_n)$ be vectors in D, and let $F : D \to D$ be the function $F(x) = (f_1(x), \ldots, f_n(x))$. The dimension of D, x, y, z, and F is n, written $d(D) = d(x) = d(y) = d(z) = d(F) = n$. I will use the following notation:

$$
\begin{aligned}
D_S &\equiv D_{s_1} \times D_{s_2} \times \cdots \times D_{s_k} \\
x_S &\equiv (x_{s_1}, x_{s_2}, \ldots, x_{s_k}) \\
(y_S, z_{\bar{S}}) &\equiv x \text{ where } x_S = y_S \text{ and } x_{\bar{S}} = z_{\bar{S}} \\
F_S(x) &\equiv (F(x))_S \\
F^{[S]}(x) &\equiv (F_S(x), x_{\bar{S}})
\end{aligned}
$$

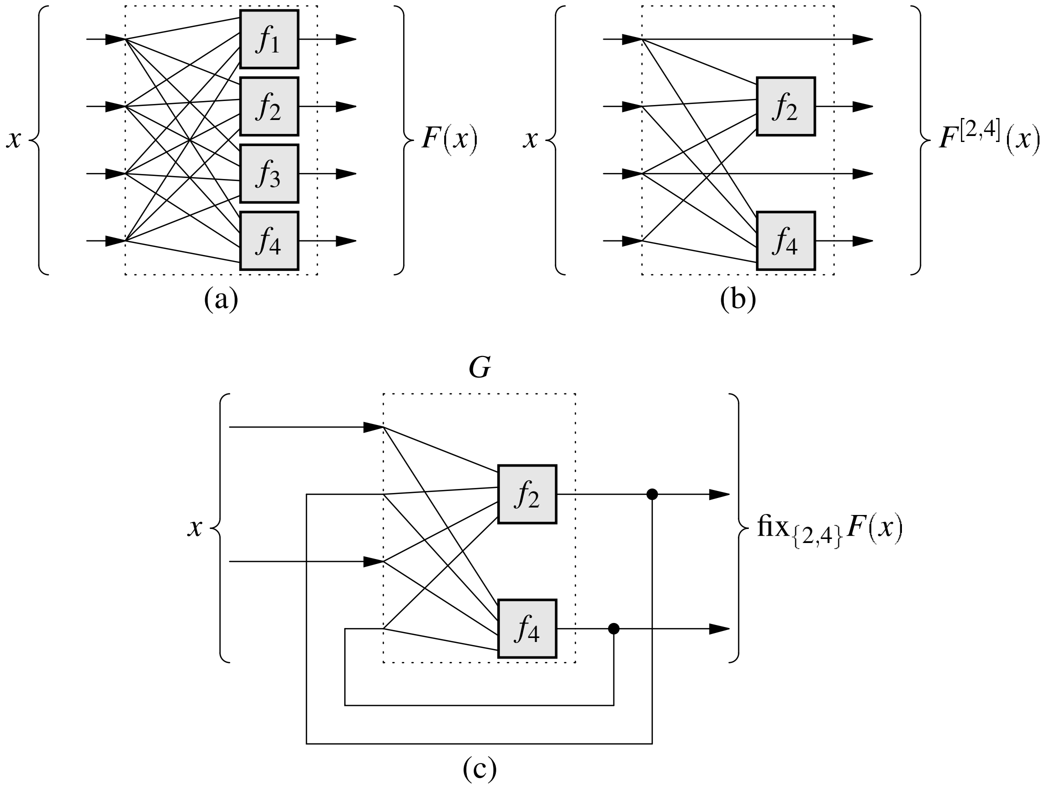


Figure 4.2 An illustration of function constructions. (a) $F(x)$ (b) $F^{[2,4]}(x)$ (c) $\mathrm{fix}_{\{2,4\}}F(x)$

$$\text{fix}_S(F(x)) \quad \equiv \quad \text{fix}(G(x_S)) \text{ where } G(x_S) = F_S(x_S, x_{\bar{S}})$$

where $x_S : D_S$, $F_S : D \to D_S$, $F^{[S]} : D \to D$, and $\text{fix}_S : (D \to D) \to D_S$. x_S, D_S, and F_S are simple projections. $F^{[S]}$ evaluates only the elements of S, leaving the others unchanged, as shown in Figure 4.2(b). fix_S is the most subtle: it is the least fixed point of part of a function where the elements of S are the variables and the others are constants. Figure 4.2(c) depicts this construction.

Ultimately, we will be finding a sequence of sets S_1, S_2, …, S_k (an iteration strategy) such that

$$F^{[S_k]}(F^{[S_{k-1}]}(\cdots(F^{[S_2]}(F^{[S_1]}(\bot)))\cdots)) = \text{fix}(F). \tag{4.3}$$

When F is a monotonic function, it turns out that all the intermediate results in such an expression satisfy a strong set of properties I call the Chaotic Iteration Invariants. The idea is that at any point, evaluating any subset of elements, i.e., $F^{[S_i]}$ can only increase the result and it can never pass the least fixed point of any *part* of the function. These properties will be essential in proving that a particular iteration strategy actually computes the least fixed point.

Definition 14 *A vector $c \in D_1 \times \cdots \times D_n$ satisfies the **Chaotic Iteration Invariants** with respect to a function $F : D_1 \times \cdots \times D_n \to D_1 \times \cdots \times D_n$ if, for all subsets $A \subseteq \{1, \ldots, n\}$,*

$$c \sqsubseteq F^{[A]}(c) \text{ and } c_A \sqsubseteq \text{fix}_A(F(c)).$$

Theorem 6 *The Chaotic Iteration Invariants hold for $c = \bot$.*

Proof Trivial, since $\bot \sqsubseteq x$ for any x. ∎

The following proof provides the inductive step to show the Chaotic Iteration Invariants hold for the intermediate results in evaluating the least fixed point (done by (4.3)). It builds the desired relations by breaking expressions into four sets of indices (AB, $A\bar{B}$, $\bar{A}B$, and $\bar{A}\bar{B}$) and considering each separately. This uses the pointwise $\sqsubseteq$ relation from Proposition 4 to ensure that if $a_{S_k} \sqsubseteq b_{S_k}$ over some set of subsets $S_1 \cup S_2 \cup \ldots = \{1, \ldots, n\}$ then $a \sqsubseteq b$.

Theorem 7 *If c satisfies the Chaotic Iteration Invariants with respect to a monotonic function F, then $d = F^{[B]}(c)$ also satisfies the invariants for all subsets $B \subseteq \{1, \ldots, n\}$.*

Proof Let $A \subseteq \{1, \ldots, n\}$ be some subset and let $e = F^{[A]}(d)$. By definition,

$$d_{\bar{A}} = e_{\bar{A}}. \tag{4.4}$$

Next, by assumption, $c \sqsubseteq F^{[B]}(c)$, and $F^{[B]}$ is monotonic, so

$$d = F^{[B]}(c) \sqsubseteq F^{[B]}(F^{[B]}(c)).$$

However, $e_{AB} = (F^{[B]}(F^{[B]}(c)))_{AB}$, so

$$d_{AB} \sqsubseteq e_{AB}. \tag{4.5}$$

Similarly, since $F^{[A]}$ is monotonic, $F^{[A]}(c) \sqsubseteq F^{[A]}(F^{[B]}(c))$. Moreover, $c \sqsubseteq F^{[A]}(c)$, so $c \sqsubseteq e$. However, $d_{A\bar{B}} = c_{A\bar{B}}$, so

$$d_{A\bar{B}} \sqsubseteq e_{A\bar{B}}. \tag{4.6}$$

Together, (4.4), (4.5), and (4.6) imply one of the chaotic iteration invariants for d, i.e.,

$$d \sqsubseteq e = F^{[A]}(d). \tag{4.7}$$

To show the other chaotic iteration invariant, first note $d_B = F_B(c)$, so

$$d_{AB} = F_{AB}(c). \tag{4.8}$$

Furthermore, since $d_{\bar{B}} = c_{\bar{B}}$, $d_{A\bar{B}} = c_{A\bar{B}}$, and since c satisfies the chaotic iteration invariant, $c_S \sqsubseteq F_S(c)$ for all S,

$$d_{A\bar{B}} = c_{A\bar{B}} \sqsubseteq F_{A\bar{B}}(c). \tag{4.9}$$

Together, (4.8) and (4.9) imply

$$d_A \sqsubseteq F_A(c). \tag{4.10}$$

Second, since c satisfies the chaotic iteration invariants,

$$c_A \sqsubseteq \mathrm{fix}_A(F(c)).$$

Applying the monotonic function $F_A(\cdot, c_{\bar{A}})$ to both sides gives

$$F_A(c_A, c_{\bar{A}}) = F_A(c) \sqsubseteq F_A(\mathrm{fix}_A(F(c)), c_{\bar{A}}).$$

The right side of this is the application of a function to its fixed point, so

$$F_A(c) \sqsubseteq F_A(\mathrm{fix}_A(F(c)), c_{\bar{A}}) = \mathrm{fix}_A(F(c)). \tag{4.11}$$

Finally, since c satisfies the chaotic iteration invariant, $c_S \sqsubseteq F_S(c)$ for all S, $c_{B\bar{A}} \sqsubseteq F_{B\bar{A}}(c) = d_{B\bar{A}}$. Moreover, since $d_{\bar{B}\bar{A}} = c_{\bar{B}\bar{A}}$, it follows that $c_{\bar{A}} \sqsubseteq d_{\bar{A}}$. Because F is monotonic, this implies

$$\mathrm{fix}_A(F(c)) \sqsubseteq \mathrm{fix}_A(F(d)) \tag{4.12}$$

Together, (4.10), (4.11), and (4.12) imply the other chaotic iteration invariant, i.e.,

$$d_A \sqsubseteq \mathrm{fix}_A(F(d)).$$

This and (4.7) show d satisfies the chaotic iteration invariants. ■

4.2.3 Series/Parallel Decomposition

The following theorem, inspired by Robert [58], shows that you arrive at the same least fixed point if you evaluate a vector-valued function in pieces. This result allows the blocks of an SR system be evaluated in-place and a whole block at a time, rather than single outputs.

Theorem 8 (Robert) *If $F : D \to D$ is a continuous function on a finite-height n-dimensional pointed CPO D and*

$$G = F^{[S_m]} \circ \cdots \circ F^{[2]} \circ F^{[1]}$$

where $S_k \subseteq \{1, \ldots, n\}$ and $S_1 \cup \cdots \cup S_m = \{1, \ldots, n\}$, then F and G have the same unique least fixed point.

Proof F has a unique least fixed point since it continuous on a pointed CPO (Theorem 2). Furthermore, since F is continuous, $F^{[S_k]}$ is, and G is (Proposition 8), so G also has a unique least fixed point (Theorem 2).

Let $x = \text{fix}(F)$. Since $F(x) = x$, it follows from Theorem 6, Theorem 7, and Theorem 2 that

$$\begin{array}{rcl}
\perp & \sqsubseteq & x \\
F^{[S_1]}(\perp) & \sqsubseteq & F^{[S_1]}(x) = x \\
(F^{[S_2]} \circ F^{[S_1]})(\perp) & \sqsubseteq & F^{[S_2]}(x) = x \\
& \vdots & \\
G(\perp) & \sqsubseteq & x \\
& \vdots & \\
(G \circ G)(\perp) & \sqsubseteq & x \\
& \vdots & \\
\text{fix}(G) & \sqsubseteq & x = \text{fix}(F)
\end{array}$$

Since D is a finite-height CPO, $\text{fix}(G) = G^{h(D)}$ (Theorem 5), so $\text{fix}(G)$ satisfies the Chaotic Iteration Invariant. Let $y = \text{fix}(G)$. It follows that

$$y \sqsubseteq F^{[S_1]}(y) \sqsubseteq (F^{[S_2]} \circ F^{[S_1]})(y) \sqsubseteq \cdots \sqsubseteq G(y) = y$$

so $F^{[S_k]}(y) = y$ for all k. Thus, y must be a fixed point of each component that appears in any S_k. Since $S_1 \cup \cdots \cup S_m = \{1, \ldots, n\}$, y is a fixed point of F, yet $y \sqsubseteq \text{fix}(F)$, the least fixed point, so it must be that $\text{fix}(F) = \text{fix}(G)$ ■

4.2.4 Partitioned Evaluation

The following theorem due to Bekić [2]* provides a way to find the fixed point of a function by partitioning it into a head (F_H) and a tail

*Bekić originally proved this in a 1969, but was not published until after his death in 1982. I take the proof from Winskel [73, Chapter 10].

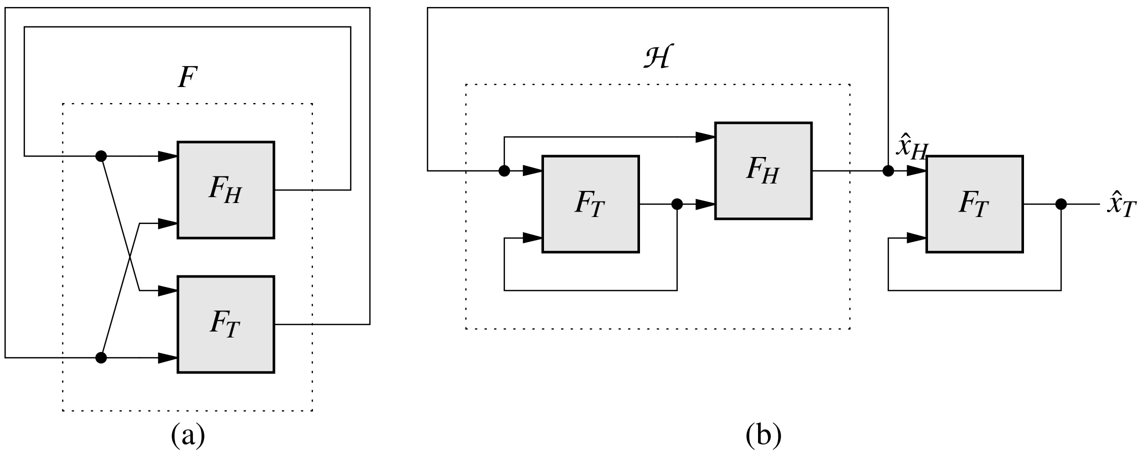


Figure 4.3 A visualization of Theorem 9. (a) The least fixed point of $F = (F_H, F_T)$. (b) The decomposition into $\mathcal{H}$ cascaded with T. $\mathcal{H}$ combined with its feedback loop is topologically identical to (a).

(F_T), finding the fixed point of $\mathcal{H}$ (the head with an embedded evaluation of the fixed point of the tail), and using this to find the fixed point of the tail, as illustrated in Figure 4.3. Evaluating a fixed point this way is more efficient when calculating the fixed point of the tail is easy or when the head does not depend on the tail.

Theorem 9 (Bekić) *If $F : D \to D$ is a continuous function on an n-dimensional pointed CPO D and $H \subseteq \{1, \ldots, n\}$, then* $\text{fix}(F) = \hat{x} = (\text{fix}(\mathcal{H}), \text{fix}(\mathcal{T}))$, *where*

$$\mathcal{H}(x_H) = F_H(x_H, \text{fix}_T(F(x_H, x_T))) \tag{4.13}$$

$$\mathcal{T}(x_T) = F_T(\hat{x}_H, x_T) \tag{4.14}$$

and $T = \bar{H}$.

Proof By definition, $\hat{x}_T = \text{fix}(\mathcal{T})$, so $\mathcal{T}(\hat{x}_T) = \hat{x}_T = F_T(\hat{x}_H, \hat{x}_T)$. Similarly, $\hat{x}_H = \text{fix}(\mathcal{H})$, so

$$\mathcal{H}(\hat{x}_H) = \hat{x}_H = F_H(\hat{x}_H, \text{fix}_T(F(\hat{x}_H, x_T))) = F_H(\hat{x}_H, \hat{x}_T),$$

so $\hat{x}$ is a fixed point of F.

Now, let y be the least fixed point of F, which exists because of Theorem 2. Since y_T is a fixed point of $F(y_H, \cdot)$, it must be approximated by the least fixed point, i.e.,

$$\text{fix}_T(F(y_H, x_T)) \sqsubseteq y_T.$$

Since F is continuous, $F_H(x_H, x_T)$ is monotonic in x_T and therefore

$$F_H(y_H, \text{fix}_T(F(y_H, x_T))) \sqsubseteq F_H(y_H, y_T).$$

The left side of this is $\mathcal{H}(y_H)$, and since $F_H(y_H, y_T) = y_H$, this implies $\mathcal{H}(y_H) \sqsubseteq y_H$, so y_H is a prefixed point of $\mathcal{H}$. From Theorem 2, it follows that

$$\text{fix}(\mathcal{H}) \sqsubseteq y_H. \tag{4.15}$$

Since $\hat{x}_H = \text{fix}(\mathcal{H})$, this implies $\hat{x}_H \sqsubseteq y_H$, and since $F_T(x_H, x_T)$ is continuous and hence monotonic in x_H, it follows that

$$F_T(\hat{x}_H, y_T) \sqsubseteq F_T(y_H, y_T).$$

Since $F_T(y_H, y_T) = y_T$, this implies y_T is a prefixed point of $\mathcal{T}$, and again by Theorem 2,

$$\text{fix}(\mathcal{T}) \sqsubseteq y_T. \tag{4.16}$$

Since y is the least fixed point, and $\hat{x}$ is a fixed point, $y \sqsubseteq \hat{x}$. However, from (4.15) and (4.16), $\hat{x}_H \sqsubseteq y_H$ and $\hat{x}_T \sqsubseteq y_T$. It follows that $\hat{x} = y$. ∎

Definition 15 *If $F : D \to D$ is an n-dimensional function and (H, T) are a pair of disjoint subsets of $\{1, \ldots, n\}$, then the partition (H, T) is* ***separable*** *if $F_H(x_H, x_T, x_{\overline{(HT)}})$, is independent of x_T, i.e., if $F_H(x_H, x_T, x_{\overline{(HT)}}) = F_H(x_H, x'_T, x_{\overline{(HT)}})$ for all x'_T.*

The idea of a separable partition is simple: its head does not depend on its tail. This makes it easier to compute the least fixed point since both halves can be evaluated in isolation. This is illustrated in Figure 4.4.

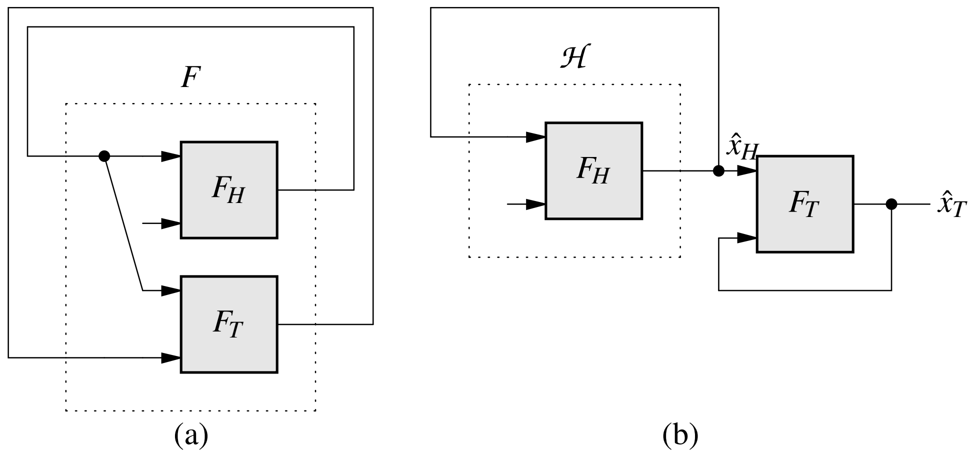


Figure 4.4 A visualization of Corollary 3. (a) The least fixed point of $F = (F_H, F_T)$, a separable partition since F_H is independent of x_T. (b) The decomposition into $\mathcal{H}$ cascaded with T.

Corollary 3 *If F is separable, then the least fixed point of f is $\hat{x} = (\text{fix}(\mathcal{H}), \text{fix}(\mathcal{T}))$, where $T = \bar{H}$,*

$$\begin{aligned} \mathcal{H}(x_H) &= F_H(x_H, y_T) \\ \mathcal{T}(x_T) &= F_T(\hat{x}_H, x_T), \end{aligned}$$

and y_T may be anything.

4.2.5 The Divide-and-Conquer Least Fixed Point Algorithm

The fixed-point algorithm, Figure 4.5, uses divide-and-conquer. It divides the problem using Bekić's Theorem (Theorem 9), and evaluates the fixed points using iterative evaluation (Theorem 5, whose conditions are ensured by the Chaotic Iteration Invariants of Definition 14). Using Bekić's Theorem can require fewer function evaluations than using iterative evaluation directly, but choosing a division point that actually reduces the number is difficult. I devote Section 4.3 to solving this problem.

$\textsc{Fix}(F, S, x)$
if S should be partitioned
  choose H s.t. $\varnothing \subset H \subset S$ — *Choose a head*
  $T = S - H$ — *The tail is the remainder*
  if (H, T) is separable — *Separable Partition:*
    $x = \textsc{Fix}(F, H, x)$ — *fixed point of the head*
    $x = \textsc{Fix}(F, T, x)$ — *fixed point of the tail*
  else — *Non-separable Partition:*
    for $i = 1, \ldots, h(F_H)$ — *Evaluate* $\mathcal{H}^{h(H)} = \text{fix}(\mathcal{H})$
      $x = \textsc{Fix}(F, T, x)$ — *fixed point of the tail*
      $x = F^{[H]}(x)$ — *evaluate the head*
    $x = \textsc{Fix}(F, T, x)$ — *Evaluate* $\text{fix}(\mathcal{T})$
else — *Iterative Evaluation:*
  for $i = 1, \ldots, h(F_S)$ — *Evaluate* $F_S^{h(F_S)}$
    $x = F^{[S]}(x)$
return x

Figure 4.5 The divide-and-conquer fixed point algorithm. It computes $(\text{fix}_S(F(x)), x_{\bar{S}})$. In particular, $\textsc{Fix}(F, \{1, \ldots, d(F)\}, \perp) = \text{fix}(F)$. When and how S is partitioned is the subject of Section 4.3.

Theorem 10 *If x satisfies the Chaotic Iteration Invariant, the algorithm in Figure 4.5 computes*

$$\textsc{Fix}(F,S,x) = (\text{fix}_S(F(x)), x_{\bar{S}}) \tag{4.17}$$

Proof This terminates because the cardinality of S decreases by at least one for each recursive call.

Only two statements modify x, i.e., $x = F^{[H]}(x)$ and $x = F^{[S]}(x)$. It follows from Theorem 7 that the Chaotic Iteration Invariant on x is maintained.

When S is not partitioned, the second loop computes

$$(F^{[S]}(x))^{h(F_S)} = (\text{fix}_S F(x), x_{\bar{S}})$$

since x satisfies the Chaotic Iteration Invariant, and hence the conditions in Theorem 5.

When S is partitioned, assume the recursive calls of $\textsc{Fix}(F,S,x)$ satisfy (4.17).

When H is separable, the first call computes $x_H = \text{fix}_H(F(x))$, and the second call computes $x_T = \text{fix}_T(F(x))$. From Corollary 3, these satisfy (4.17).

When H is not separable, the first loop computes $x_H = \mathcal{H}^{h(F_H)} = \text{fix}(\mathcal{H})$ by Theorem 5, then uses this to compute $x_T = \text{fix}(\mathcal{T})$. From Theorem 9, these also satisfy (4.17). ■

4.3 Devising Efficient Schedules

The divide-and-conquer algorithm in Figure 4.5 will find the least fixed point regardless of when and where the function is partitioned, but says nothing about how this should be done. In this section, I use this freedom to improve the quality of the schedules that come from recording this algorithm's behavior.

Minimizing the worst-case execution time of a schedule is the primary objective because SR systems assume the synchrony hypothesis. The worst-case time limits the minimum time between successive events because to correctly model synchronous behavior, an SR

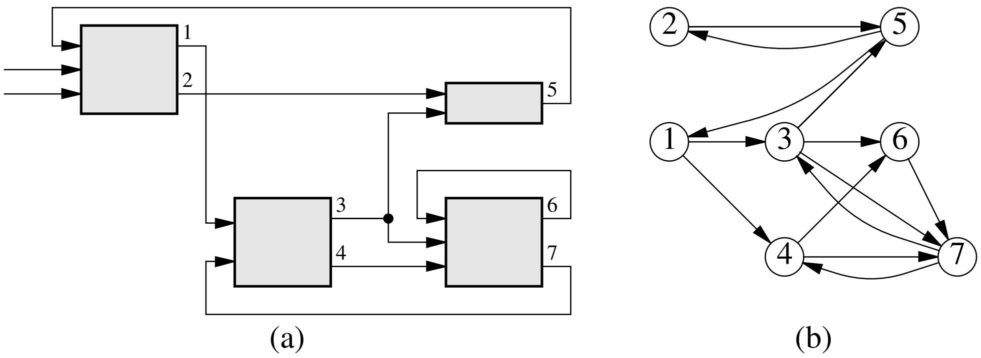


Figure 4.6 (a) The system from Figure 3.6. (b) Its dependency graph.

system must finish computing before more inputs arrive. A scheme that improves the average or best-case execution time at the expense of the worst-case is of no use for system running in real-time.

More efficient schedules might be possible if detailed information about the functions or possible data values in the system were known. However, the assumption of heterogeneity limits what can be known about the functions, and consequently, also limits knowledge of what data values might appear.

Having discounted the possibility of function- or data-dependent partitioning schemes, only the communication structure remains to select the partitioning scheme. Fortunately, this turns out to be an effective way to find fast schedules.

I introduce the dependency graph, an abstraction of the communication structure of a system. It is a directed graph with a vertex for each communication channel (or equivalently, block output). There is an edge from each block's input channels to all its outputs, indicating functional dependence or information flow. The edges are essentially the connections of Definition 10 (Page 49). Figure 4.6 shows a dependency graph.

Definition 16 *A **directed graph** (or **digraph**) G is a pair (V,E) where V is a set of vertices and E is a set of edges. An **edge** is an element of $V \times V$ with*

(7 . (5 . 2 1 3)^1 4 6)^1

Figure 4.7 A schedule for the dependency graph in Figure 4.6.

*distinct vertices. A vertex v_n is **reachable** from v_1 if there is a **path** from v_1 to v_n: a set of edges such that*

$$\{(v_1, v_2), (v_2, v_3), (v_3, v_4), \ldots, (v_{n-1}, v_n)\} \subseteq E.$$

Definition 17 *The **dependency graph** $G = (V, E)$ of an n-dimensional function F has*

$$\begin{aligned} V &= \{v_1, \ldots, v_n\} \\ E &= \{\, (v_j, v_k) \mid \textit{if } f_k(x) \textit{ depends on } x_j \textit{ and } j \neq k \,\}. \end{aligned}$$

The fixed communication structure in an SR system leads naturally to a recursive partitioning strategy. The unchanging structure means choosing a single way to partition a given subgraph can be optimal, suggesting a simple recursive decomposition of the graph. When the least fixed point algorithm is invoked on a particular subgraph, the choice of whether to partition, and if so, how, is the same each time. The schedule contains exactly this information.

The syntax I adopt for my schedules follows naturally from the recursive fixed point algorithm. There are three cases. I enclose a non-separable partition in parentheses, writing it (*head* . *tail*)^n, where n is the height of the head. The halves of a separable partition are just juxtaposed, e.g., *head tail*. I denote the evaluating of a single node with a number, and I enclose multiple nodes to be evaluated in brackets. Figure 4.7 shows a schedule for the graph in Figure 4.6. I formally define the syntax in Backus-Naur form:

$$\begin{array}{rcll} s & \to & i & \text{Evaluate the } i\text{th component alone} \\ & | & [\ i_1 \cdots i_k\ ] & \text{Evaluate } i_1, \ldots, i_k \text{ in parallel} \\ & | & s_1\ s_2 & \text{Evaluate } s_1 \text{ then } s_2 \\ & | & (\ s_1\ .\ s_2\ )^n & \text{Non-separable partition with} \\ & & & \text{head } s_1 \text{ and tail } s_2 \end{array}$$

The meaning of a schedule is a function built from the composition of a series of function evaluations. I define a schedule's meaning using a denotational style.* The function $\mathcal{E} : \mathit{Sched} \to (D \to D)$ transforms a schedule (a syntactical object) into a function. Expressions within double brackets are written in the syntax of schedules. Note that function composition reads right-to-left.

$$\begin{array}{rcl} \mathcal{E}[\![\ i\ ]\!] & = & F^{[i]} \\ \mathcal{E}[\![\ [\ i_1 \cdots i_k\ ]\ ]\!] & = & F^{[i_1,\ldots,i_k]} \\ \mathcal{E}[\![\ s_1\ s_2\ ]\!] & = & \mathcal{E}[\![\ s_2\ ]\!] \circ \mathcal{E}[\![\ s_1\ ]\!] \\ \mathcal{E}[\![\ (\ s_1\ .\ s_2\ )^n\ ]\!] & = & \mathcal{E}[\![\ s_2\ ]\!] \circ (\mathcal{E}[\![\ s_1\ ]\!] \circ \mathcal{E}[\![\ s_2\ ]\!])^n \end{array}$$

For example, the meaning of the schedule in Figure 4.7 is derived as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{E}[\![(5\ .\ 2\ 1\ 3\ )^1\ 4\ 6]\!]} \\ = & F^{[6]} \circ F^{[4]} \circ \mathcal{E}[\![(5\ .\ 2\ 1\ 3\ )^1]\!] \\ = & F^{[6]} \circ F^{[4]} \circ F^{[3]} \circ F^{[1]} \circ F^{[2]} \circ F^{[5]} \circ F^{[3]} \circ F^{[1]} \circ F^{[2]} \\ \multicolumn{2}{l}{\mathcal{E}[\![(\ 7\ .\ (\ 5\ .\ 2\ 1\ 3\ )^1\ 4\ 6\ )^1]\!]} \\ = & \mathcal{E}[\![(5\ .\ 2\ 1\ 3\ )^1\ 4\ 6]\!] \circ (F^{[7]} \circ \mathcal{E}[\![(5\ .\ 2\ 1\ 3\ )^1\ 4\ 6]\!])^1 \\ = & F^{[6]} \circ F^{[4]} \circ F^{[3]} \circ F^{[1]} \circ F^{[2]} \circ F^{[5]} \circ F^{[3]} \circ F^{[1]} \circ F^{[2]} \circ F^{[7]} \circ \\ & F^{[6]} \circ F^{[4]} \circ F^{[3]} \circ F^{[1]} \circ F^{[2]} \circ F^{[5]} \circ F^{[3]} \circ F^{[1]} \circ F^{[2]} \end{array}$$

This is a simple recursive interpretation that corresponds directly to the behavior of the algorithm in Figure 4.5. A single number corresponds to evaluating that component of the function. Juxtaposed

*Such notation is standard in the denotational semantics community. See a standard text such as Winskel [73].

schedules, which must be separable, are taken one after the other. Non-separable partitions are evaluated iteratively.

Each node appears exactly once in a schedule. This is a direct consequence of the recursive nature of optimal partitioning, which splits a set into disjoint sets.

4.3.1 The Minimum Evaluation Cost

In this section, I characterize minimum-cost schedules. I show their cost always falls between linear and quadratic (acyclic systems have linear cost, fully-connected systems have quadratic cost). Another result, that greedily taking separable partitions is optimal, seems obvious, but is difficult to prove. Finding good schedules becomes much easier since identifying separable partitions is easy. The final result also provides a useful insight: an optimal non-separable partition must have a separable tail. Later, this will allow me to find good partitions more efficiently since it restricts what sort of partition I should look for.

To characterize schedules, I make two assumptions about the functions and their domains. First, I assume each wire's CPO is flat—each has a height of one. Interpreted another way, each wire is either undefined ($= \perp$), or has a value. This assumption greatly simplifies analysis and is reasonable for most applications. The main consequence of this is that the height and dimension of a function (or CPO) are equal, i.e.,

$$h(F) = d(F).$$

I also assume the cost of evaluating any output is constant. This greatly simplifies analysis, since the cost of evaluating a function is just its dimension, but it can be an oversimplification. Especially in heterogeneous systems, the cost of evaluating blocks can vary significantly. However, many of the results arising from this assumption can be translated to systems where this assumption is relaxed.

At each level, the divide-and-conquer algorithm takes one of three

Type	Cost
Separable Partition	$C_s(H,T) = C(H) + C(T)$
Non-Separable Partition	$C_p(H,T) = d(H)^2 + (d(H)+1)C(T)$
Iterative Evaluation	$C_i(S) = d(S)^2$

Table 4.1 The cost of execution strategies compared. $C(S)$ is the minimum cost of evaluating the fixed point of S. All CPOs are of height one, and the cost of evaluating a function is its dimension.

approaches, whose cost I discuss below and summarize in Table 4.1. $C(S)$ denotes the minimum cost of evaluating $\mathrm{fix}_S(F(x))$. I extend the $d(\cdot)$ notation to include sets, i.e., $d(S)$ denotes the dimension of S—its cardinality.

Separable Partition When (H,T) is a separable partition, the least fixed points of the head and tail are each evaluated once. The cost is

$$C_s(H,T) = \begin{cases} C(H) + C(T) & \text{if } H \cap T = \varnothing \text{ and} \\ & (H,T) \text{ is separable} \\ \infty & \text{otherwise.} \end{cases}$$

Non-Separable Partition When (H,T) is non-separable, the head is evaluated $h(H)$ times and the least fixed point of the tail is evaluated $h(H)+1$ times (once outside the loop). The cost is

$$C_p(H,T) = \begin{cases} d(H)^2 + (d(H)+1)C(T) & \text{if } H \cap T = \varnothing \\ \infty & \text{otherwise.} \end{cases}$$

Iterative Evaluation It takes $h(S)$ evaluations of $F^{[S]}$ to compute its fixed point iteratively, so the cost of this is

$$C_i(S) = d(S)^2.$$

The minimum cost of evaluating the least fixed point of a function, then, is the least cost among all separable partitions, all non-separable partitions, and iterative evaluation.

$$C(S) = \min_{\varnothing \subset H \subset S} \{C_s(H, S-H) \cup C_p(H, S-H) \cup C_i(S)\}$$

The next two theorems provide tight bounds on the optimal cost of finding the least fixed point. It falls between linear, which corresponds to evaluating the whole function once, and quadratic, corresponding to evaluating the whole function as many times as it has outputs. Besides providing insight into the overall cost of running SR systems, these bounds will be used to speed up the branch-and-bound scheduling algorithm I present in Section 4.3.3.

Theorem 11 *The cost of evaluating the least fixed point of S is at least its dimension, i.e.,*

$$C(S) \geq d(S). \tag{4.18}$$

Moreover, it is possible that $C(S) = d(S)$*, so the bound is tight.*

Proof I will show (4.18) by induction on $d(S)$.

For $d(S) = 1$, the function can only be evaluated iteratively, so $C(S) = C_i(S) = d(S)^2 = 1$. This satisfies (4.18).

For $d(S) > 1$, there are three possibilities. For iterative evaluation,

$$C(S) \geq C_i(S) = d(S)^2 > d(S).$$

For separable partitions,

$$C(S) \geq C_s(H, S-H) = C(H) + C(S-H) \geq d(H) + d(S-H) = d(S).$$

And for non-separable partitions,

$$\begin{aligned} C(S) \geq C_p(H, S-H) &= d(H)^2 + (d(H)+1)C(S-H) \\ &\geq d(H)^2 + (d(H)+1)d(S-H) \\ &= d(H)(d(H) + d(S-H)) + d(S-H) \\ &= d(H)d(S) + d(S-H) \\ &\geq d(S). \end{aligned}$$

The case $C(S) = d(S)$ occurs when there exists separable partitions at each level of the recursion. ∎

Theorem 12 *The minimum cost of evaluating the least fixed point of S satisfies*

$$C(S) \leq d(S)^2 - (d(S) - 1). \tag{4.19}$$

Moreover, when no separable partitions exist,

$$C(S) = d(S)^2 - (d(S) - 1),$$

corresponding to evaluating a partition where $d(S-H) = 1$, so the bound is tight.

Proof I will show this by induction on $d(S)$. For $d(S) = 1$, $C(S) = C_i(S) = 1$, satisfying (4.19).

When $d(S) > 1$, the least fixed point of S can always be evaluated as a non-separable partition where $d(S-H) = 1$. Since $C(S-H) = 1$ in this case, the overall cost satisfies

$$\begin{aligned} C(S) \leq C_p(H, S-H) &= (d(H))^2 + (d(H)+1)C(S-H) \\ &= (d(S)-1)^2 + d(S) \\ &= d(S)^2 - (d(S)-1). \end{aligned}$$

If no separable partitions are possible, $C(S) = d(S)^2 - (d(S) - 1)$. Since I have shown (4.19), I will prove this by showing through induction on $d(S)$ that

$$C(S) \geq d(S)^2 - (d(S) - 1). \tag{4.20}$$

For $d(S) = 1$, $C(S) = C_i(S) = 1$, which satisfies (4.20).

For $d(S) > 1$, there are two possibilities. For the iterative evaluation case, $C_i(S) = d(S)^2$, and for the non-separable partition case,

$$\begin{aligned} C_p(H, S-H) &= d(H)^2 + (d(H)+1)C(S-H) \\ &= x^2 + (x+1)C(S-H) \\ &\geq x^2 + (x+1)((n-x)^2 - (n-x-1)) \\ &= n^2 - (n-1) + c \end{aligned}$$

where $c = x(x-(n-1))(x-(n-2))$. Since this term is non-negative for the valid values of $x = d(H)$, i.e., $d(H) = 1, 2, \ldots, n-1$, it follows that $C_p(H, S-H) \geq d(S)^2 - (d(S)-1)$.

Since by assumption either $C(S) = C_i(S)$ or $C(S) = C_p(H, S-H)$. Both are greater than the right side of (4.20), so it follows that (4.20) holds. ∎

Corollary 4 *Unless $d(S) = 1$, iterative evaluation is non-optimal, i.e., $C(S) < C_i(S)$.*

The next theorem, in effect, says that greedily evaluating separable partitions is optimal. This is not a very surprising result, but is fairly tedious to prove.

Theorem 13 *When possible, evaluating a separable partition is optimal, i.e., if $(H, S-H)$ is separable, then*

$$C(S) = C_s(H, S-H). \tag{4.21}$$

Proof I will show this through induction on $d(S)$.

First, consider the case where $d(S) = 2$. Let $(H, S-H)$ be separable. Since $d(H) = d(S-H) = 1$, $C_s(H, S-H) = 1+1 = 2$. The cost of any non-separable partition is $C_p(H, S-H) = 1^2 + (1+1) \cdot 1 = 3$. The cost of iterative evaluation is $C_i(S) = 2^2 = 4$. Thus for $d(S) = 2$, (4.21) holds.

Now consider $d(S) = k$ for some $k > 2$. Assume (4.21) holds for all $d(S) < k$. From Theorem 12, it follows that $C(S) < C_i(S)$, so either $C(S) = C_s(H', S-H')$ or $C(S) = C_p(H', S-H')$ for some H'. For convenience, write $T = S-H$ and $T' = S-H'$.

Consider $C_p(H', T')$. The partition (HT', TT') of T' is separable since (H, T) is separable (see Figure 4.8, and recall juxtaposition denote set intersection). Hence by the inductive assumption,

$$C(T') = C_s(HT', TT') = C(HT') + C(TT'). \tag{4.22}$$

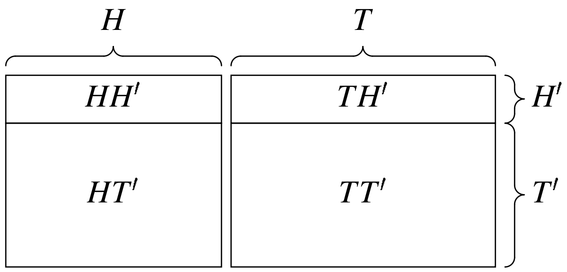


Figure 4.8 A visualization for part of Theorem 13. Here (H,T) is separable and the non-separable (H',T') is assumed to be less costly.

Certainly H and T can each be evaluated as non-separable partitions (HH',HT') and (TH',TT') respectively (see Figure 4.8), so it follows that

$$\begin{aligned} C(H) &\leq d(HH')^2 + (d(HH')+1)C(HT') \\ C(T) &\leq d(TH')^2 + (d(TH')+1)C(TT') \\ C(H)+C(T) &\leq d(HH')^2 + d(TH')^2 + \\ &\quad (d(HH')+1)C(HT') + \\ &\quad (d(TH')+1)C(TT'). \end{aligned} \tag{4.23}$$

However, since $d(HH')+d(TH') = d(H')$ and both quantities are positive,

$$d(HH')^2 + d(TH')^2 \leq d(H')^2. \tag{4.24}$$

Furthermore, from (4.22) and since $d(H') = d(HH')+d(TH')$, it follows that

$$(d(HH')+1)C(HT') + (d(TH')+1)C(TT') \leq (d(H')+1)C(T'). \tag{4.25}$$

This can be seen visually in Figure 4.9. Together, (4.23), (4.24), and (4.25) imply

$$\begin{aligned} C(H)+C(T) &\leq d(H')^2 + (d(H')+1)C(T') \\ C_s(H,T) &\leq C_p(H',T') \end{aligned}$$

So no non-separable partition can be evaluated more efficiently.

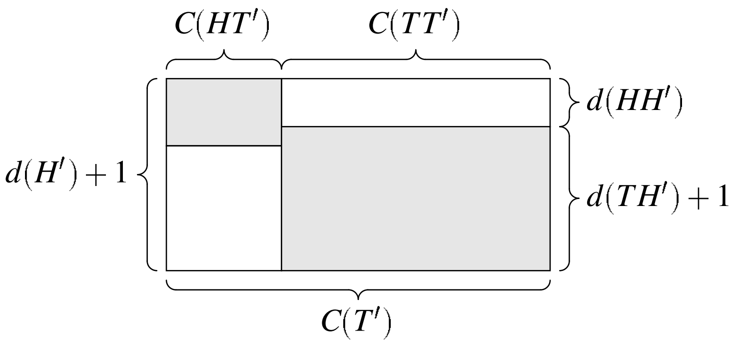


Figure 4.9 A graphical argument for (4.25). The area of the enclosing box corresponds to the left hand side; the area of the two shaded boxes corresponds to the right hand side.

Now assume there exists another separable partition (H',T'). It follows (see Figure 4.10) that

$$\begin{array}{ll} (HH',TH') = H' & (HH',HT') = H \\ (HT',TT') = T' & (TH',TT') = T \end{array}$$

are separable. By the inductive assumption, this implies

$$\begin{aligned} C(H') &= C(HH') + C(TH') \\ C(T') &= C(HT') + C(TT') \\ C(H) &= C(HH') + C(HT') \\ C(T) &= C(TH') + C(TT'). \end{aligned}$$

Taken together, these imply $C_s(H,T) = C_s(H',T')$. Thus, all separable partitions have the same cost.

Since a separable partition is always less costly than iterative evaluation or evaluating any non-separable partition, and all separable partitions have the same cost, it follows that evaluating any separable partition is optimal. ∎

Certain non-separable partitions cost more than simple-minded iterative evaluation. This occurs when

$$d(H)^2 + (d(H)+1)C(T) \quad > \quad d(S)^2$$

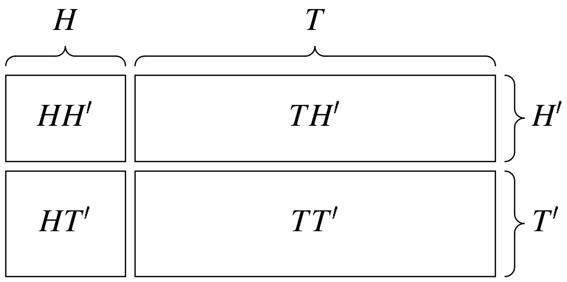


Figure 4.10 A visualization for part of Theorem 13. Here both (H,T) and (H',T') are separable.

$$\begin{aligned} C(T) &> \frac{d(S)^2 - d(H)^2}{d(H)+1} \\ &= d(T)\frac{d(S)+d(H)}{d(H)+1}. \end{aligned}$$

If $C(T) = d(T)$ this inequality is not satisfied, but for larger values of $C(T)$, it can be.

This next theorem is very useful because it restricts the number of partitions that must be considered when looking for optimal non-separable partitions. The result is not surprising: an optimal non-separable partition must have an easier-to-evaluate tail.

Theorem 14 *If a non-separable partition is optimal, then its tail must have a separable partition. I.e., if $S = (H,T)$ is optimal and if T has no separable partition, then $C(S) \leq C_p(H,T)$. The inequality is strict unless $d(S-H) = 2$.*

Proof Assume T has no non-separable partition and $C(S) = C_p(H,T)$, it follows that $C(T) = C_p(H', T-H')$ for some partition $H' \subset T$. Let $x = d(H)$, and $y = d(H')$. It follows that

$$\begin{aligned} C(S) = C_p(H,T) &= x^2 + (x+1)C(T) \\ &= x^2 + (x+1)(y^2 + (y+1)C(T-H')) \\ &= x^2 + xy^2 + y^2 + (x+1)(y+1)C(T-H') \\ &= x^2 + xy^2 + y^2 + (x+y+1+xy)C(T-H') \end{aligned}$$

$$= \quad x^2 + y^2 + (x+y+1)C(T-H') + xy(y + C(T-H')).$$

However, consider the cost of placing both H and H' in the partition. Note that $d(H+H') = x+y$.

$$\begin{aligned} C_p(H+H', T-H') &= (x+y)^2 + (x+y+1)C(T-H') \\ &= x^2 + 2xy + y^2 + (x+y+1)C(T-H') \\ &= x^2 + y^2 + (x+y+1)C(T-H') + 2xy \end{aligned}$$

When $d(T) = 2, C(T-H') = y = 1$, and $C_p(H,T) = C_p(H+H', T-H')$ since $xy(y + C(T-H')) = 2xy = 2$. For larger values of $y = d(H')$ or $C(T-H')$, $C(T-H') \geq 2$, $C_p(H+H', T-H') < C_p(H,T)$, so $C(S) < C_p(H,T)$. ■

4.3.2 Finding Good Partitions

The last section showed that optimal schedules contain separable partitions. The results section show that non-separable partitions corresponds to strongly connected components (SCCs) in the dependency graph, and show how to break an SCC into a separable partition. Together, these lead to an optimal scheduling algorithm that decomposes the dependency graph into SCCs and recurses on each component after removing a set of vertices (the head) that breaks its strong connectivity. This follows because Theorem 13 implies a separable partition is always optimal when it exists, and it turns out a decomposition of a graph into strongly connected components is unique.

Definition 18 *A digraph $G = (V,E)$ is **strongly connected** if for all vertices $v_j \neq v_k$, v_k is reachable from v_j.*

Definition 19 *A digraph $G = (V,E)$ is **acyclic** if, for all vertices $v_j \neq v_k$, if v_k is reachable from v_j, then v_j is not reachable from v_k.*

Definition 20 *Let $G = (V, E)$ be a digraph, and let $S \subseteq V$ be a set of vertices. The **border** of S is the set of all vertices outside S with an edge from S, i.e.,*

$$\text{border}_G(S) = \{ v \mid v_s \in S,\ v \in V - S,\ \textit{and}\ (v_s, v) \in E \}.$$

Definition 21 *A **kernel** of a digraph $G = (V, E)$ is a strict, non-empty subset of vertices $T \subset V$ with no border, i.e., such that* $\text{border}_G(T) = \varnothing$.

Theorem 15 *A digraph is not strongly connected if and only if it has a kernel.**

Proof (If) Assume T is a kernel of G, and let $H = V - T$. Since it is a kernel, there is no edge from any vertex in T to any vertex in H (both sets are non-empty, so this is not vacuous), so there cannot be a path from any vertex in T to any vertex in H. Hence the graph is not strongly connected.

(Only If) Assume $G = (V, E)$ is not strongly connected. It follows that there is a vertex v_h unreachable from another vertex v_t. Let T be those vertices in V reachable from v_t, including v_t, and let $H = V - T$. There cannot be an edge from a vertex in T to a vertex in H since the vertex in H would then be reachable from v_t, a contradiction. It follows, since T is non-empty (it includes v_t) and a strict subset of V (it excludes v_h), that T is a kernel of G. ■

Theorem 16 *A function has a separable partition if and only if its dependency graph has a kernel.*

Proof (If) Assume T is a kernel of a dependency graph G, and let $H = V - H$. By definition, there is no edge from any vertex in T to any vertex in H. It follows from the definition of a dependency graph that $F_H(x_H, x_T)$ is independent of x_T.

*I took the term "kernel" and this theorem from Frank [27], who describes them as "well-known," but I know of nothing else that uses this term or contains this proof.

(Only If) Assume (H,T) is a separable partition. Since $F_H(x_H, x_T)$ is independent of x_T, there cannot be any edge from a vertex in T to a vertex in H. By definition, T is a kernel. ■

Thus a separable partition is isomorphic to a kernel of the dependency graph, and these only occur when the graph is not strongly connected.

Definition 22 *The **strongly connected component decomposition** of a digraph $G = (V,E)$ is a partition of the vertex set V such that all pairs of vertices in a single partition are mutually reachable. Each partition is a **strongly connected component** or **SCC**.*

Fortunately, a graph can be decomposed into its strongly connected components in linear time using a well-known algorithm due to Tarjan [67].

The primary challenge, then, is to partition a strongly connected dependency graph. Theorem 14 implies no optimal partition leaves the tail strongly connected.

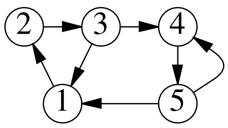
Choosing a head that leaves the tail acyclic would satisfy this. An acyclic graph is clearly not strongly connected, and furthermore, it is the least expensive to evaluate since the SCCs of an acyclic graph are all single vertices—it can be evaluated in linear time.

However, a partition that produces an acyclic graph may not be optimal, as illustrated by the system in Figure 4.11. Removing $\{1,4\}$, a minimal set whose removal leaves the graph acyclic, does not lead to an optimal schedule.

Not only can the minimum feedback vertex set lead to sub-optimal schedules, identifying it is an NP-complete problem.* Clearly, this is not the best solution.

Finding a partition that leaves the graph acyclic is too strong—breaking strong connectivity is enough. However, finding a minimum set of vertices that breaks strong connectivity can be done in

*See Garey and Johnson [28], the standard reference for these problems.



$([1\ 4]\ .\ 2\ 3\ 5)^2$ uses a minimum feedback vertex set and has cost 13

$(\ 1\ .\ 2\ 3\ (\ 4\ .\ 5\ )^1)^1$ does not use a minimum feedback vertex set and has cost 11

Figure 4.11 A system where partitioning using a minimum feedback vertex set is not optimal.

polynomial time,* unlike the Minimum Feedback Vertex Set problem. Unfortunately, using the minimum set of vertices that breaks strong connectivity is not always optimal.

The following theorem provides a simple way to break strong connectivity: pick a set of vertices and remove its border. Figure 4.12 illustrates this.

Theorem 17 *Let $G = (V, E)$ be a strongly connected digraph. The graph that results from removing H, i.e.,*

$$G' = (V', E') = (V - H, \{\, (v_a, v_b) \mid (v_a, v_b) \in E \text{ and } v_a, v_b \notin H \,\}),$$

is not strongly connected if and only if $\mathrm{border}_G(K) \subseteq H \subset V$ *and* $K \subset V - H$ *for some non-empty subset* $K \subset V$.

Proof (If) Assume $\mathrm{border}_G(K) \subseteq H \subset V$. By definition, $V' \cap \mathrm{border}_G(K) = \varnothing$ since $V' = V - H$, so $\mathrm{border}_{G'}(K) = \varnothing$ and K is a kernel since $K \subset V - H$. From Theorem 15, G' is not strongly connected.

*Kuller [40] pointed this out to me. The basic idea is that the maximum flow between any two points in a network with vertex capacities of one is the minimum number of vertices that must be removed to break all paths between the two points. Since to break strong connectivity, there must be at least two points between which there are no paths, the minimum overall number must be the minimum flow between all pairs of vertices. Finding the maximum flow can be done using one of the polynomial-time network flow algorithms after splitting all vertices into an incoming and outgoing vertex, placing an edge with unit capacity between them, and setting all other vertex capacities to infinity.

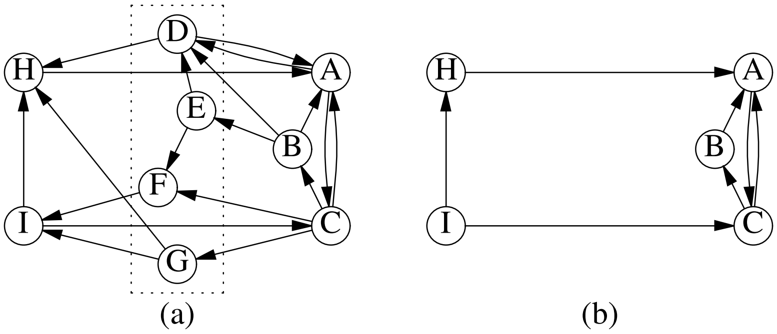


Figure 4.12 Removing a border to break strong connectivity. (a) The border of A, B, and C are all vertices with edges coming from A, B, and C. (b) Removing this border (vertices D, E, F, G) breaks this graph's strong connectivity.

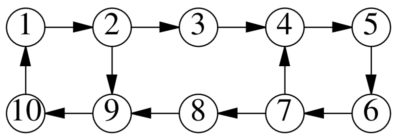


Figure 4.13 An example where a valid partition $\{3\}$ is not a predecessor or successor set of any single vertex.

(Only If) Assume G' is not strongly connected. By Theorem 15, it must have a kernel K, i.e., $\text{border}_{G'}(K) = \varnothing$. For this to be the case, the border of K in G must have been removed, i.e., $\text{border}_G(K) \subseteq H$. ■

Corollary 5 *Removing the successor or predecessor set of any vertex in a strongly connected graph breaks its strong connectivity.*

Proof The successor set of a vertex v is exactly $\text{border}_G(\{v\})$, and its predecessor set is $\text{border}_G(V - \{v\})$. Both sets satisfy the conditions in Theorem 17. ■

Corollary 5 provides one way to find partitions that break strong connectivity, but not all valid partitions are the predecessor or successor of any vertex. Figure 4.13 depicts an example where a minimal vertex set is not of this form.

4.3.3 The Branch and Bound Algorithm

The results in the last two sections suggest a recursive branch-and-bound strategy for finding the optimal schedule. Theorem 13 suggests greedily using separable partitions is optimal, and Theorems 15 and 16 imply the SCCs of a graph are exactly the separable partitions. Strongly connected components must be evaluated as non-separable partitions, and Theorems 14 and 17 imply the optimal choice of head must break the tail's strong connectivity, requiring the partition to contain the border of some group of vertices.

Figure 4.14 shows an algorithm based on this strategy. It recursively decomposes a graph into strongly connected components and searches for a good partition of each.

The algorithm first uses Theorem 12 to compute a gross (quadratic) bound on the cost using the size of each strongly connected component. The bound B is forced to be no greater than this to avoid considering any grossly sub-optimal partitions.

The algorithm next considers each strongly connected component. First, a bound b on the cost of the SCC is computed by subtracting the minimum possible cost of the remaining SCCs (just their size according to Theorem 11) from the remaining cost r. Next, unless the SCC is trivial (one-dimensional), the algorithm considers some set of heads. For each head, the optimum cost of evaluating the tail is calculated by calling COST recursively. The bound for the tail comes from noting that to achieve the bound on S_k, the cost of the tail must satisfy $C_p(H, S_k - H) = d(H)^2 + (d(H)+1)C(S_k - H) \leq b$, the bound on the SCC. Solving this for $C(H, S_k - H)$ yields

$$C(H, S_k - H) \leq \left\lfloor \frac{b - d(H)^2}{d(H)+1} \right\rfloor .$$

COST(S, B)
    Decompose S into strongly connected components $S_1, \ldots, S_n$
    $B = \min\{B, \sum_{k=1}^{n} d(S_k)^2 - (d(S_k) - 1)\}$ *Bound to be met*
    $r = B$ *Remaining cost*
    foreach strongly connected component $S_1, \ldots, S_n$
        $b = r - \sum_{i=k+1}^{n} d(S_i)$ *Bound for this SCC*
        if $b < d(S_k)$
            return ∞ *Bound is impossible to meet*
        if $d(S_k) = 1$
            $r = r - 1$ *One-dimensional function case*
        else
            $a = \infty$ *Minimum achieved for this SCC*
            foreach head H of S_k
                $t = \text{COST}\left(S_k - H, \left\lfloor \frac{b - d(H)^2}{d(H) + 1} \right\rfloor\right)$ *Optimal tail cost*
                $a = \min\{a, d(H)^2 + (d(H) + 1)t\}$ *Cost including the head*
                if $a < b$
                    $b = a - 1$ *Beat it by at least one next time*
            $r = r - a$
    if $r > 0$
        return $B - r$ *Bound was met*
    else
        return ∞ *Bound was not met*

Figure 4.14 A branch-and-bound algorithm for finding the optimal partition. S is the subgraph to be partitioned, and B is the cost bound. The algorithm returns the best cost or ∞ if the bound could not be met. See Section 4.3.4 for a discussion of which heads of S_k are considered.

4.3.4 Choosing the Head of an SCC

Which heads to consider and the order in which to consider them for each strongly connected component is key to the branch-and-bound algorithm. Considering all possible heads would be correct, but Theorem 14 implies this is overkill—only those partitions that leave the tail separable can possibly be optimal. Theorem 17 provides a way to construct any partition that breaks strong connectivity.

The minimum cost of a particular partition grows rapidly with the size of the head, providing a way to quickly discount large heads. From Theorem 11, it follows that a partition worth considering must satisfy

$$C_p(H, S_k - H) = d(H)^2 + (d(H)+1)C(F_k - H) \leq b,$$

and $C(S_k - H) \geq d(S_k - H)$, it follows that $d(H)$ must satisfy

$$\begin{aligned} d(H)^2 + (d(H)+1)(d(S_k - H)) &\leq b \\ d(H)^2 + (d(H)+1)(d(S_k) - d(H)) &\leq b \\ d(H)^2 + d(H)d(S_k) + d(S_k) - d(H) - d(H)^2 &\leq b \\ d(H) &\leq \frac{b - d(S_k)}{d(S_k) - 1}. \end{aligned}$$

This result also suggests taking the partitions in order of increasing $d(H)$. This will tighten the bound faster since the cost grows at least as quickly as $d(H)^2$.

The best heuristic I have found for reducing the number of heads to consider comes from Theorem 17, which implies an optimal head (i.e., one that breaks strong connectivity) contains a border set. Starting with each vertex in the graph, the heuristic "grows" a vertex set by greedily adding the vertex in its border that increases the size of the border the least. Since the graph is strongly connected, there is always at least one vertex in any border. The heads considered are the borders of these vertex sets.

For example, running this on the graph in Figure 4.13 produces, for the set that begins with vertex 10,

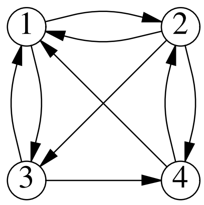
Vertex Set	Border/Partition Considered
10	1
10 1	2
10 1 2	3 9
10 1 2 9	3
10 1 2 9 3	4
10 1 2 9 3 4	5
10 1 2 9 3 4 5	6
10 1 2 9 3 4 5 6	7

I also tried using Corollary 5 to only consider one partition: the smallest outset (vertices with edges coming from) or inset (vertices with edges leading to) of any vertex. Frequently, no schedule based on these partitions would meet the worst-case quadratic bound—the best schedules would be worse than using a simple brute-force approach.

I also tried simply considering all single vertex partitions. This is frequently sub-optimal since for many strongly connected graphs, the removal of a single vertex does not break strong connectivity. (See the example in Figure 4.15.) Again, it was often the case that for certain systems, the cost of any schedules based on these partitions exceeded the quadratic upper bound.

4.3.5 Schedule Transformations

SR systems are composed of blocks that can only be evaluated as a whole, but the schedules produced by the branch-and-bound algorithm evaluate a single output at a time. Simply evaluating a whole block any time a single output needs to be evaluated is correct (this corresponds to introducing more evaluations, which Theorem 8 says will not affect the result), but wasteful.



$([\ 1\ 2\ ].\ 3\ 4\ )^2$ is an optimal schedule with cost 10.

$(1\ .\ (\ 4\ .\ 2\ 3\ )^1)^1$ contains only single-vertex heads and has cost 11.

Figure 4.15 A network whose optimal schedule has no single-vertex heads.

In this section, I present an algorithm for restructuring a schedule to minimize the number of redundant block evaluations. The idea is simple: combine outputs on a particular block into a parallel section of the schedule by moving them past sections on which they do not depend. To facilitate this, I have identified five rewrite rules that can restructure a schedule without affecting its correctness. They depend on two main results. The first is that a part of a schedule can be moved before a section whose results on which it does not depend. The second is that introducing additional function evaluations does not affect correctness (follows from Theorem 8).

I present the rewrite rules in a deductive style. The subexpression above the line can be replaced with the subexpression below if the predicate to the right is satisfied.

I introduce the following two functions to describe what inputs a subexpression depends on and what outputs it affects. Both take a schedule an return a set of indices. $O[\![\ s\ ]\!]$ is simply the list of all indices that appear in the subexpression, while $I[\![\ i\ ]\!]$ is simply the vertices with edges going to i in the dependency graph.

$$
\begin{aligned}
I[\![\ s\ ]\!] &= \{\, i \mid \mathcal{E}[\![\ s\ ]\!](x) \text{ depends on } x_i \,\} \\
O[\![\ s\ ]\!] &= \{\, i \mid \mathcal{E}[\![\ s\ ]\!](x)_i \neq x_i \text{ for some } x \,\}
\end{aligned}
$$

$$\frac{s\ i}{i\ s} \quad \text{when } I[\![\ i\ ]\!] \cap O[\![\ s\ ]\!] = \varnothing \tag{4.26}$$

$$\frac{(\ s_1\ .\ s_2\ )^n\ i}{(\ s_1\ .\ s_2\ i\ )^n} \quad \text{always} \tag{4.27}$$

$$\frac{(\ s_1\ .\ s_2\ )^n\ i}{(\ s_1\ i\ .\ s_2\ )^n} \quad \text{when } I[\![\ i\ ]\!] \cap O[\![\ s_2\ ]\!] = \varnothing \tag{4.28}$$

$$\frac{(\ i\ s_1\ .\ s_2\ )^n}{(\ s_1\ .\ s_2\ i\ )^n} \quad \text{always} \tag{4.29}$$

$$\frac{i_1 \cdots i_n}{[\ i_1 \cdots i_n\ ]} \quad \text{when } \forall j < k\ .\ O[\![\ i_j\ ]\!] \cap I[\![\ i_k\ ]\!] = \varnothing \tag{4.30}$$

The first rule, (4.26), allows an index to be moved before a sub-schedule when it is independent. The second, (4.27), allows a later output to be moved into a tail. This works because the two sequences are

$$s_2\ s_1\ s_2\ \cdots\ s_1\ s_2\ i$$
$$s_2\ i\ s_1\ s_2\ i\ \cdots\ i\ s_1\ s_2\ i.$$

The bottom just has additional i's. Similarly, (4.28) allows a later output to be moved into a tail. The sequence on the bottom is

$$s_2\ s_1\ i\ s_2\ \cdots\ s_1\ i\ s_2$$

which has moved i to the left of s_2 and added others. The two sequences in (4.29) are

$$s_2\ i\ s_1\ s_2\ i\ \cdots\ i\ s_1\ s_2$$
$$s_2\ i\ s_1\ s_2\ i\ \cdots\ i\ s_1\ s_2\ i.$$

which differ only by the addition of a trailing i. When none of the outputs of a group affect any later evaluations, the group can be evaluated in parallel, leading to (4.30).

The rules in (4.26)–(4.30) suggest an algorithm for merging block evaluations. The objective is to merge outputs on the same block into a single parallel evaluation. (4.26)–(4.28) suggest an index i can be pushed lexicographically toward the front of the tail until one of its inputs is encountered. An output in a head can first be moved to the end of its tail with (4.29), then pushed toward the front of its tail. Once two or more independent outputs on the block are pushed together, they can be merged into a bracket-enclosed parallel evaluation block using (4.30).

Merge(s)
  for each output i in s
    Determine i's "pushable range"
    if an output on the same block appears in the range
      Find the leftmost acceptable position for i.
      Merge i with the nearest output on the same block to the right, if any.

Figure 4.16 An algorithm for merging outputs on the same block.

The heads of non-separable partitions can be restructured freely. The divide-and-conquer algorithm produces a parallel execution of all the outputs in the head, but they can be serialized arbitrarily (from Theorem 8). In particular, each block with an output appearing in the head can be executed exactly once, and in any order.

Figure 4.17 shows an example of this algorithm's behavior.

4.3.6 Experimental Results

To test the efficiency of the branch-and-bound algorithm presented in Figure 4.14 and the partition selection heuristics in Section 4.3.4, I generated 304 SR systems at random and found the minimum cost schedule for each using the branch-and-bound algorithm using two algorithms for choosing SCC partitions. My exact algorithm considers all partitions containing one vertex, then all partitions containing two vertices, etc. My "sweep" heuristic only considers a subset of all possible partitions (and as such often misses the optimal schedule) by growing a vertex set starting from each vertex in the graph, as described in Section 4.3.4.

To create the random examples, I generated sixteen systems with two blocks, sixteen with three blocks, etc., up to twenty blocks. For each block in a system, I randomly selected a number of inputs and outputs, each between zero and ten (uniformly distributed), and then

$(7 . (5 . 2\ 1\ 3)^1\ 4\ 6)^1$

(a)

Output	Pushable Range	Leftmost Position
1	$(7 . (5 . \underline{2}\ 1\ 3)^1\ 4\ 6)^1$	$(7 . (5 . \bullet\ 2\ 1\ 3)^1\ 4\ 6)^1$
2	empty	
3	$(7 . (5 . [\ \underline{2\ 1}\ ]\ 3)^1\ 4\ 6)^1$	no matching block (4)
4	$(7 . \underline{(5 . [2\ 1]\ 3)^1}\ 4\ 6)^1$	$(7 . (5 . [2\ 1] \bullet 3)^1\ 4\ 6)^1$
5	$(7 . (5 . \underline{[2\ 1]\ [3\ 4]})^1\ 6)^1$	no matching block
6	$(7 . \underline{(5 . [2\ 1]\ [3\ 4])^1}\ 6)^1$	no matching block (7)
7	$(7 . \underline{(5 . [2\ 1]\ [3\ 4])^1\ 6})^1$	$(7 . (5 . 2\ 1\ 3)^1\ 4\ 6 \bullet)^1$

(b)

$(7 . (5 . [2\ 1]\ [3\ 4])^1\ 6)^1$

(c)

Figure 4.17 (a) The schedule from Figure 4.7. (b) The effects of running the algorithm in Figure 4.16. (c) The final merged schedule.

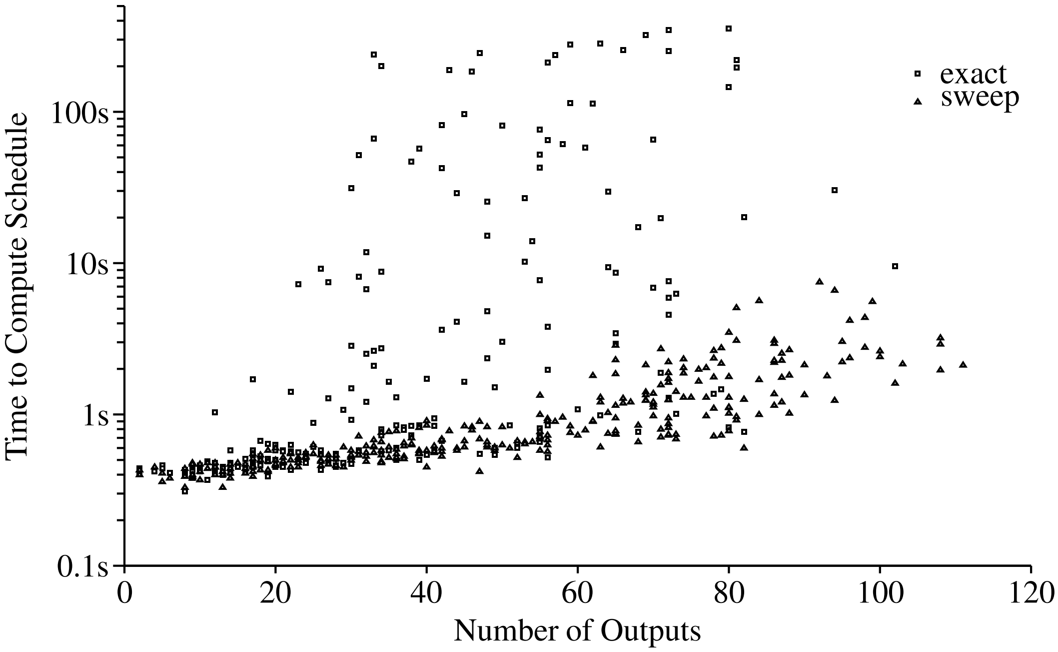


Figure 4.18 A comparison of scheduling times for the branch-and-bound algorithm using the exact and heuristic sweep partition generators. All times are on a SPARCStation 10.

for each block's input, randomly chose a block and an output port on the block to connect to. If the block I chose had no outputs, I left the input unconnected.

For reference, all data were collected on a a SPARCStation 10 with 96MB of main memory, although the program never consumed more than about 4MB. All times include the time to initialize the program and load the system, typically a few hundred milliseconds.

Figure 4.18 shows the times it took the branch-and-bound algorithm to compute the schedule for each system using the exact and sweep heuristics. The number of outputs in the system is plotted horizontally (the sum of the number of outputs on each block—exactly the number of vertices in the dependency graph). The times are plotted vertically on a logarithmic scale. The exact algorithm required over 500 seconds to compute a schedule for 98 systems (out of 304), but the sweep heuristic always completed in under eight seconds.

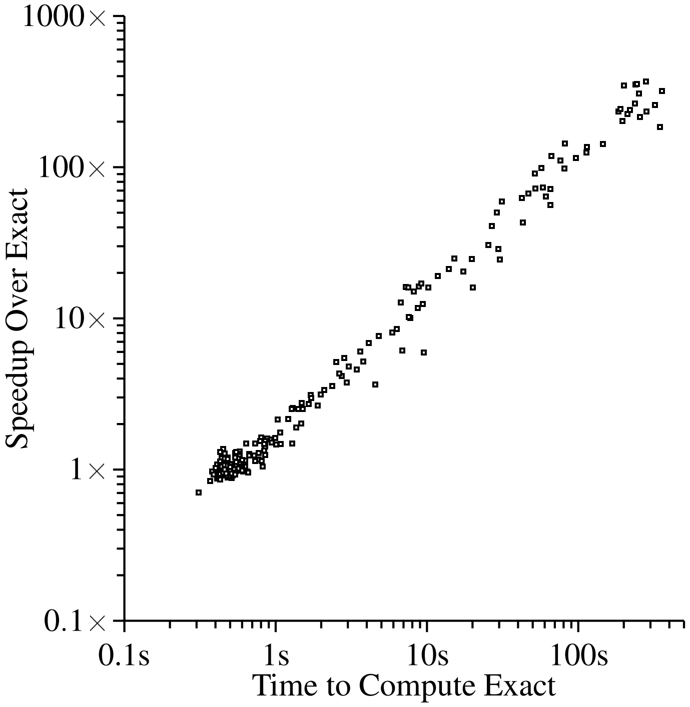


Figure 4.19 The scheduling time speedup arising from using the heuristic sweep partition generator.

From Figure 4.18, it appears the time to run the exact heuristic varies substantially and grows quickly. The time it takes to run the sweep heuristic does appear to be growing exponentially, but very slowly. Moreover, the time for the heuristic seems much more predictable in comparison.

Figure 4.19 shows the sweep heuristic is exponentially more efficient than the exact brute-force solution. Although the speedup is between $1\times$ and $2\times$ about 40% of the time, and the heuristic is actually slower in about 20% of the cases, this is only the case when both the exact and heuristic times are fairly small. For longer times (e.g., one second or more), the heuristic partitioner is the clear winner by an exponentially growing margin.

To save time, the heuristic partitioner considers only a subset of all possible partitions. Unfortunately, it can miss the optimal partition, leading to the cost increases shown in Figure 4.20, but these are not

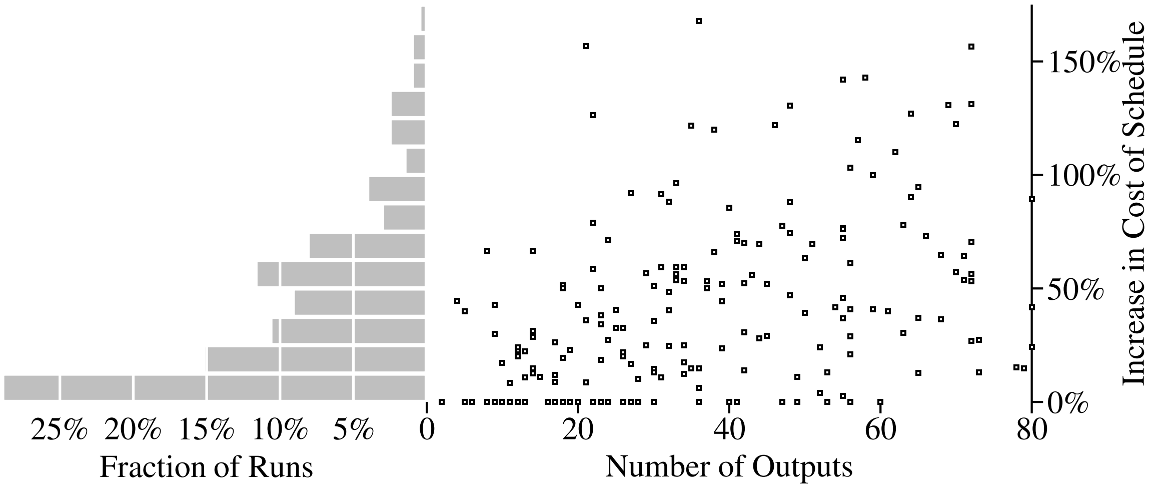


Figure 4.20 The increase in schedule cost from using the heuristic sweep partition generator.

awful. The increase is less than 12% for more than an quarter of the cases. Interestingly, the cost increase does not appear to be related to the problem size.

Theorem 11 says the minimum schedule cost must be at least the number of vertices in the dependency graph (i.e., the total number of outputs in the system), and Theorem 12 says it must be less than quadratic. The graph in Figure 4.21 bears this out—the cost of all schedules falls between the n and n^2 lines. However, more interestingly, the asymptotic bound appears to be closer to $n^{1.5}$. Of course, this a function of the systems I chose to schedule, and there are systems whose optimal schedule costs $n^2-(n-1)$, but there do not appear to be many of them. Moreover, since the random graph construction algorithm I presented above produces something reasonably close to real systems, I expect similar results for real systems.

From these results, I conclude that both the exact and heuristic partitioning schemes have merit. In many cases, finding the exact answer is computationally feasible, but when it is not, the heuristic scheme is far faster and produces comparable results—half of the time within 25% of the optimal schedule, and rarely more than twice as bad.

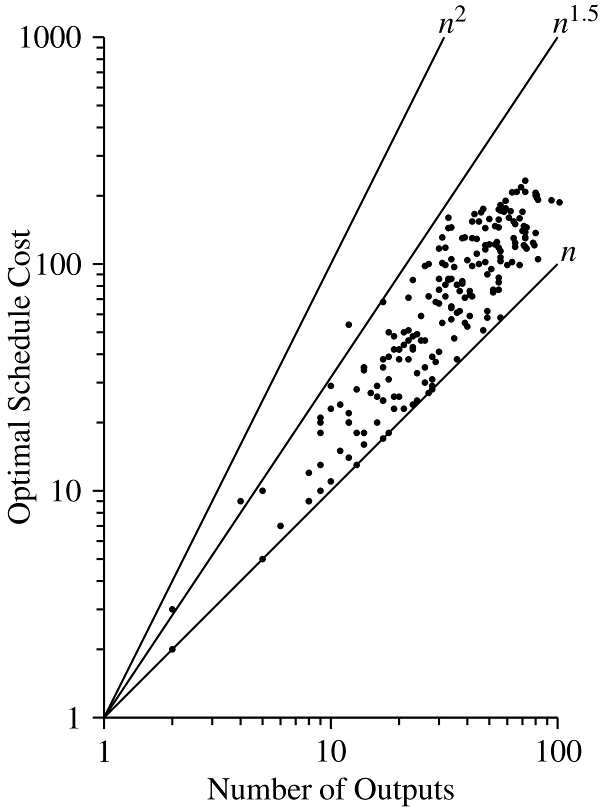


Figure 4.21 The minimum schedule cost as a function of graph size.

Chapter 5

Implementation

The management question, then,
is not whether *to build*
a pilot system and throw it away.
You will *do that.*
The only question is whether to plan in advance
to build a throwaway, or to promise to deliver
the throwaway to customers.
—F. P. Brooks, *The Mythical Man-Month*

I IMPLEMENTED the SR model of computation in Ptolemy [16, 17], an environment for heterogeneous system prototyping. Here, I present the details of how I did this, along with two sizable examples that demonstrate how the SR model of computation can be used to specify reactive systems.

The ideas in Ptolemy, particularly its view of heterogeneity, were a driving force behind this research. A variety of synchronous languages have been proposed (see Section 2.3), but none explore the problem of how to assemble heterogeneous systems. The solution I devised followed naturally from the Ptolemy philosophy.

The two examples I present illustrate two different applications of the SR model of computation. The first system, an electronic address book, is dominated by its user interface. The second, a MIDI synthesizer, is a real-time heterogeneous system that uses a different model of computation to implement some of its behavior.

5.1 Ptolemy

Ptolemy [16, 17] is an object-oriented environment for simulating and synthesizing embedded systems. Ptolemy is written in C++ [66] and uses the Tcl/Tk language [53] for some user interface duties.

Ptolemy describes its systems as block diagrams. A system is a collection of blocks and connections between input and output ports on those blocks. The blocks may come from one of the existing libraries, users may write their own in C++ or some other language, or a block may contain another block diagram, allowing for hierarchical designs.

To implement the SR model of computation, I created a new simulation domain in Ptolemy. A Ptolemy domain is an embodiment of a particular model of computation, and each consists of a set of blocks that conform to the model and a scheduler responsible for determining an execution order for the blocks in a system. For example, a block in the Synchronous Dataflow (SDF) domain (see Section 2.4.2) produces and consumes a fixed number of data tokens from communication FIFOs each time it executes. A block in my Synchronous Reactive (SR) domain examines its inputs and writes a value, "absent," or "unknown" on each output.

In many Ptolemy domains, including my SR domain, all scheduling decisions can be made before the system is run. Such static scheduling reduces overhead since the run-time scheduler may simply read off a list of blocks to fire. It also enables a style of software synthesis known as block code generation, in which code for each block is simply inlined in the order prescribed by the schedule. This reduces scheduling overhead to almost nothing—the program counter of the processor effectively functions as the run-time scheduler.

In addition to minimal run-time overhead, static scheduling allows the system to be analyzed in more detail. For example, static SDF

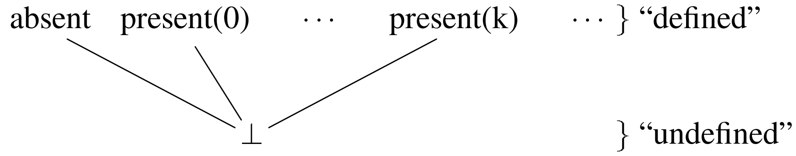


Figure 5.1 The CPO for the three states of a communication channel: unknown, absent (no event), and present (valued event). Here, the present events are integer-valued, but they could be anything.

schedulers are able to determine a bound on the size of each communication buffer, allowing faster fixed-length buffers to be used. The static SR scheduler is able to determine the per-tick execution time of the system (if it knows each block's execution time), allowing the synchrony hypothesis to be tested without resorting to extensive simulation.

5.1.1 The SR Domain

The SR domain in Ptolemy simulates systems described with the SR model of computation. The blocks in the SR domain communicate among themselves with events sent through single-driver, multiple-receiver channels. In each instant, a channel can either have an event with a value, the absence of an event, or be undefined, generally due to contradictory feedback. Communication is instantaneous and unbuffered, so each block connection (port) on a channel sees the same event (or absence thereof) in an instant. The three states of a communication channel are ordered as shown in Figure 5.1.

Blocks in SR systems must behave monotonically* to obey the semantics in Chapter 3. This means that when an SR block is given more-defined inputs (i.e., one or more inputs have changed from undefined to present or absent), switching from undefined to defined is the only way an output is allowed to change. Switching from present to absent or changing a value is prohibited.

*More precisely, they must compute continuous functions. However, since the values on the communication channels form a flat CPO, there are no infinite chains and it follows from Proposition 7 that monotonicity is sufficient.

I have written two schedulers for the SR domain. The default is a static scheduler based on the algorithms presented in Chapter 4. The other is dynamic, executing a system's blocks in essentially a random order until no outputs change, indicating the system has converged. From the results in Chapter 3, it can be shown that this algorithm correctly simulates a system.

5.1.2 SR Blocks in C++

A new C++ block for the SR domain is written by creating a new class that inherits from an existing block class and overrides certain methods. The scheduler, which calls these methods, expects them to perform functions such as updating the block's outputs and changing its state. The C++ programming interface is summarized in Table 5.1.

To create a new block, a designer writes a `.pl` file. Figure 5.2 is a simple example, Figures 5.13–5.15 on Pages 126–128 is a more complex example. describing its interface and the C++ code for certain methods. The `ptlang` preprocessor digests this file and generates C++ source and header files describing the block, which are then compiled and linked into the Ptolemy system.

Communication to and from an SR domain block goes through instances of the `InSRPort` and `OutSRPort` classes. Each represents a connection to a communication channel, and are typically declared as public data members of a block class, each with a name and a type. The state of both input and output ports may be tested with the `known()`, `present()`, and `absent()` methods, and if an event is present, its value may be read with the `get()` method.

Output ports have additional methods for changing their state. A valued event can be emitted on a formerly-undefined port by calling the `emit()` method and filling in the value of the returned Particle, a Ptolemy class describing a piece of data. Depending on the declared type of the port, this value might be an integer, a string, or a floating-

	Block Methods To Be Overridden
`void setup()`	Configure the block prior to simulation. Call methods such as `reactive()` and `independent()` here.
`void begin()`	Reset the state of the block—called once at the beginning of a simulation.
`void go()`	Update the outputs based on the inputs and state. For non-strict stars, this may be called more than once an instant and should not change the state.
`void tick()`	For non-strict stars, advance the block's state for the next tick based on its inputs and outputs. Called exactly once at the end of each instant.
	Block Methods To Be Called
`void reactive()`	Mark this block as reactive—require at least one present input before calling `go()`. Call only in the `setup()` method.
	Input/Output Port Methods
`void independent()`	Mark the input as independent—not affecting any outputs in the current instant. Call only in the `setup()` method.
`int known()`	TRUE if the port's state is not undefined
`int present()`	TRUE if the port has an event this instant
`int absent()`	TRUE if the port has no event this instant
`Particle & get()`	When `present()` returns TRUE, this returns a particle representing the value of the event. Calling this any other time is an error.
	Output-Specific Methods
`Particle & emit()`	Mark the port as having an event this instant. The returned particle should be set to the emitted value. Only call this when `known()` would return FALSE.
`void makeAbsent()`	Mark this port as having no event this instant. Only call this when `known()` would return FALSE.

Table 5.1 C++ interfaces to SR domain stars and portholes.

```
defstar {
  name { Pre }
  domain { SR }
  derivedFrom { SRNonStrictStar }

  input {
    name { input }
    type { int }
  }

  output {
    name { output }
    type { int }
  }

  state {
    name { theState }
    type { int }
    default { "0" }
    desc { Initial output value, state afterwards. }
  }

  setup {
    input.independent();
  }

  go {
    if ( !output.known() ) {
        output.emit() << int(theState);
    }
  }

  tick {
    if ( input.present() ) {
      theState = int(input.get());
    }
  }
}
```

Figure 5.2 The SR domain delay block, which delays its input by exactly one instant. The `ptlang` program translates this into C++ source and header files.

point number. The absence of an event in the current instant can be declared by calling the `makeAbsent()` method. All output ports are set to undefined at the beginning of an instant, and because of monotonicity, there is never any need for a block to reset a port to the undefined state.

Strict Blocks

An SR block is strict by default, meaning it will only be executed if all of its inputs are defined. This guarantees monotonicity and makes these blocks behave like those in most other Ptolemy domains. Writing a new block like this amounts to writing a `go()` method that updates a block's outputs and its state for the next instant. There are no restrictions about the outputs a particular set of inputs may produce.

Such a strict SR block may also be marked as reactive, which further requires at least one input to be present. If all the inputs are absent, all outputs will be marked as absent and `go()` will not be called. This further simplifies coding since many blocks in the SR domain are often reactive in this sense.

Non-Strict Blocks

The problem with strict blocks is that they do not work well in feedback loops. A feedback loop containing nothing but strict blocks will deadlock (all their outputs will remain undefined) because each block will be waiting for the others. The alternative is to write a non-strict block that is able to produce some outputs even when some inputs remain undefined.

The ability to partially evaluate outputs requires splitting output calculation and state updates into two methods, `go()` and `tick()`, because the SR schedulers may evaluate the outputs of a non-strict block multiple times within an instant to resolve the channels in feedback loops. In general, it is impossible to predict how many times

```
go() {
  if output 1 is unknown
    if the inputs are enough to decide on output 1's value
      emit output 1 or make output 1 absent
  if output 2 is unknown
    ⋮
  ⋮
  if output n is unknown
    ⋮
}
```

Figure 5.3 A standard idiom for writing the `go()` method of non-strict blocks. The schedulers guarantee that once an output is decided upon, the inputs will only become more defined.

`go()` may be called, so its function must not change until `tick()` is called.

A non-strict block must behave monotonically, restricting the outputs that may be produced by a particular set of inputs. Specifically, the `go()` method must compute a monotonic function of its inputs, meaning that if it is called with more-defined inputs (i.e., one or more have switched from undefined to either present or absent) it may either leave its outputs untouched or change some of its undefined outputs to defined, either present or absent. In particular, it may not change an output back to undefined or switch an event's value.

Schedulers for the SR domain guarantee a block's inputs follow a monotonically increasing sequence during execution in an instant.*
This simplifies the task of ensuring monotonicity, since it means that each output can be assigned a known value exactly once during the series of evaluations in an instant. This suggests the form for the `go()` method of a block shown in Figure 5.3.

*This follows from Theorem 7, which shows the Chaotic Iteration Invariants are preserved when blocks are evaluated.

Block Methods To Be Overridden	
`go`	Update the outputs based on the inputs and state. This may be called more than once an instant and should not not change the state of the block.
`tick`	Update a block's state based on its inputs and outputs. Called exactly once at the end of each instant.
Block Method for Input/Output Ports	
`read` *`port`*	Returns "`unknown`," "`absent`," or a string representing the present value on the port.
Block Method for Output Ports only	
`write` *`port value`*	Writes *`value`* to the named output port. This is interpreted as a string to emit unless *`value`* is "`absent`." To ensure monotonicity, this should not be called unless `read` would return "`unknown`."

Table 5.2 Itcl interfaces to SR domain stars and portholes.

5.1.3 SR Blocks in Itcl

SR blocks can also be described using McClennan's `[incr Tcl]` (Itcl) language [50], an object-oriented extension of Ousterhout's Tcl language [53] that facilitates rapid development and graphical user interfaces. One of its main benefits is access to the Tk toolkit, a high-level interface to windows, buttons, and so forth.

I designed the Itcl interface as a simplification of the the C++ interface. All Itcl stars are non-strict; the behavior of the ports and the `go()` and `tick()` methods remain the same.

Writing a new Itcl block amounts to creating a new Itcl class that inherits from `SRItclStar` and overrides its `go` and `tick` methods. Figure 5.4 shows an Itcl specification of a latch.

```
class SRLatch {
    inherit SRItclStar

    constructor {} {
        set state "0"
    }

    method go {} {
        if { [read output] == "unknown" } {
            write output $state
        }
    }

    method tick {} {
        set input [read input]
        if { $input != "unknown" &&
             $input != "absent" } {
            set state $input
        }
    }

    variable state
}
```

Figure 5.4 An Itcl specification of the delay block in Figure 5.2.

5.1.4 SR Blocks from Other Languages

An SR block can also be a system described in a different Ptolemy domain. It appears as a strict, reactive block that is executed at most once in an instant and only when all of its inputs are known and at least one is present.

Such embedding is done with a Ptolemy structure called a Wormhole. When a design contains a system described in a different domain, that design appears as an SR Wormhole—an SR block connected to the foreign system. The ports on this Wormhole block are special—they translate the SR communication protocol to and from a universal protocol based on single-entry buffers. Since the Wormhole is a strict block, it runs the enclosed system only when all of its input ports are defined. Present events are copied into their universal buffers, and ports without events leave their buffers empty. After the foreign system has run, output buffers containing a single piece of data appear as valued events, and empty output buffers appears as the absence of an event.

5.2 A Digital Address Book

In this section, I present a "virtual prototype" of a digital address book specified in the SR domain to illustrate a user interface application. This is intended to model a small hand-held system and provides the ability to test the user interface before building the system. Shown in Figure 5.5, it consists of a small (ten-character) alphanumeric display that displays names and phone numbers above a small keyboard. In browsing mode, pressing a key with a letter displays the first name starting with that letter, and pressing an arrow key scrolls through the names alphabetically. In editing mode, the arrow keys move an editing cursor and the remaining keys change the character beneath it. Pushing the Edit key switches between editing and browsing.

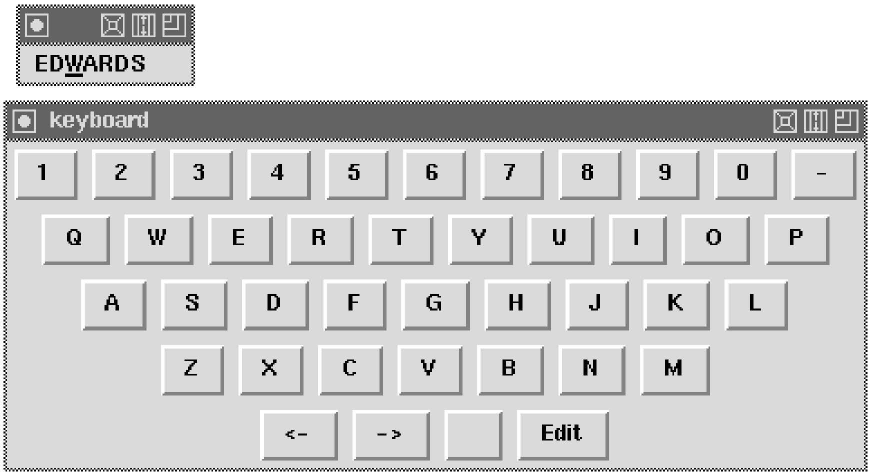


Figure 5.5 The Digital Address Book Interface. "EDWARDS" is currently being edited.

This example is fairly primitive because the language used to specify the blocks, Itcl, is not very elegant for describing finite-state behavior. It is, however, very quick to write and test.

The block diagram of the address book is shown in Figure 5.6. All blocks were custom-designed in Itcl and all communication is via string-valued events. The keyboard sends keys to the ModeSelect block, which then routes them to either the Database (responsible for storing the names) or to the Editor (responsible for controlling the cursor and modifying the name entries). The latch maintains index of the name being displayed or edited, and the counter is used to scroll through the entries while browsing.

The Keyboard Block

When a key is pressed, this emits *output* with the label on the key, and sets it absent otherwise.

To create this, I first wrote a more general Itcl keyboard class inherited from the `itk::Toplevel` class, meaning an instance of it appears as an isolated window as shown in Figure 5.5. Its constructor takes a list of keycaps and builds a button for each. The ac-

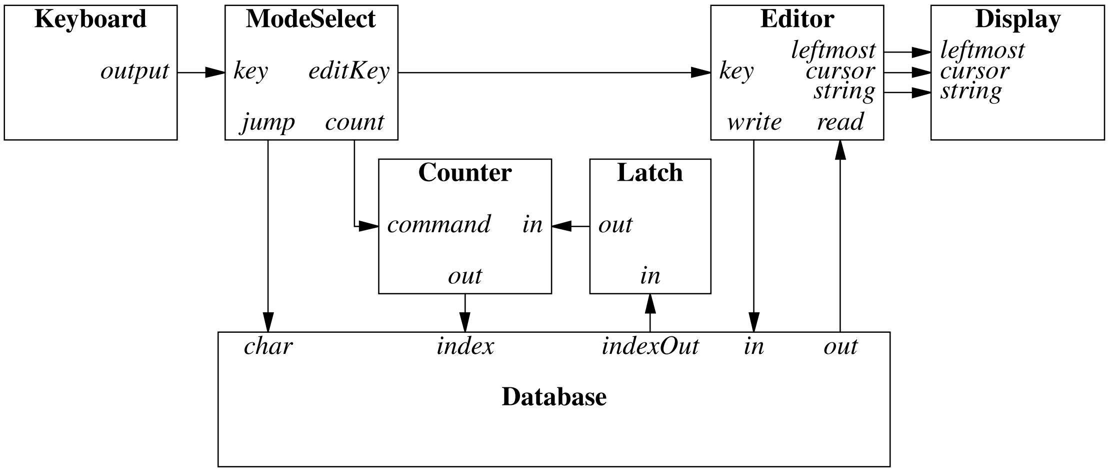


Figure 5.6 A block diagram of the Digital Address Book.

tual keyboard class inherits this and the SRItclStar class and overrides the `press` method, called whenever a key is pressed. Its `go` method emits the currently-pressed key on the output and resets the currently-pressed key variable.

The ModeSelect Block

This switches between editing and browsing modes when it receives "Edit." In browse mode, alphabetic key events are copied to *jump*, and the arrow keys emit Down or Up on *count*. Key events are copied to *editKey* in edit mode. When switching from editing to browsing, "Write" is emitted on *editKey*. When changing from either mode, *count* is "Hold."

This is essentially a large state machine implemented with a switch statement in the `go` method. Each case writes values to each output and sets the nextState variable appropriately. The `tick` method copies nextState to the current state variable.

The Counter Block

When *command* is "Up," "Down," or "Hold," *out* is respectively one more, one less, or the same as *in*. All this is implemented with a switch statement in the `go` method.

The Latch Block

Emitted on *out* is the value of *in* in the last instant. The code for this simple block is shown in Figure 5.2. It is a non-strict block since *out* is emitted without regard to the current state of *in*. This breaks the feedback loop including the latch, counter, and database by introducing a delay.

The Database Block

The most complex of the blocks, the database maintains a sorted list of strings. An event on *char* causes the database to search for the first entry beginning with that character and emit its index on *indexOut*, its value on *out*. If both *index* and *in* have an event, the string value on *in* is written into the list at the index value, the database is sorted, and the new index is emitted on *indexOut*. If an event arrives on *index* and *in* is absent, a string is fetched from the database and emitted on *out*; its index emitted on *indexOut*.

The code of the database block is shown in Figures 5.7 and 5.8.

The Editor Block

This block maintains the string being edited, which is read and written through *read* and *write*. It also maintains the current location of the edit cursor, emitted through *cursor*, and an index of the leftmost character to be displayed, emitted through *leftmost*.

When the key input is "Edit" and *read* is present, *read* becomes the string to edit. When *key* is an alphanumeric character, the character

```
# class SRDatabase
#
# INPUTS char - Selects the first item in the list starting with this
#        index - The requested index value
#        in - A string to write into index
# OUTPUTS indexOut - The current index out
#         out - The string stored at the current index
class SRDatabase {
    inherit SRItclStar
    constructor {} {
        set contents "Ann Beth Carol Debby Elizabeth Francine "
    }
    method go {} {
        set index [read index]
        set char [read character]
        set in [read in]
        if { [read out] == "unknown" &&
            $index != "unknown" && $char != "unknown" } {
                if { $index == "absent" && $char == "absent" } {

                # Both index and char absent--don't respond
                write out absent
                write indexOut absent
            } else {
                if { $char != "absent" } {

                    # char present--search for the first entry
                    set length [llength $contents]
                    for {set index 0} {$index < $length} {incr index} {
                        if { [lindex $contents $index] >= $char } {
                            break;
                        }
                    }
                    write out [lindex $contents $index]
                    write indexOut $index
                } else {
```

Figure 5.7 Itcl code for the Database block of the Digital Address Book, first part

```
                    if { $in != "unknown" } {
                        if { $in != "absent" } {

                            # index and in present--insert new value,
                            # sort the database, and find the entry
                            if { $index >= [llength $contents] } {
                                lappend contents $in
                                set contents [lsort $contents]
                            } else {
                                set contents [lsort \
                                  [lreplace $contents $index \
                                    $index $in] ]
                            }
                            set index [lsearch $contents $in]
                            write out $in
                            write indexOut $index
                        } else {

                            # index present, in absent--fetch
                            # the value from the database
                            if { $index < 0 } {
                                set index [llength $contents]
                            }
                            if { $index > [llength $contents] } {
                                set index 0
                            }
                            write out [lindex $contents $index]
                            write indexOut $index
                        }
                    }
                }
            }
        }
    }
    # A list containing the database entries
    variable contents
}
```

Figure 5.8 Itcl code for the Database block of the Digital Address Book, last part at the cursor position is overwritten and the cursor moved right.

This block is non-strict to break the feedback loop involving the database. The value of the *write* output is based only on the *key* input, not on *read*, which is dependent on the value of *in*.

The Display Block

This displays the value of the most-recent event on *string*, starting from the *leftmost* index. When *cursor* is present, the character at its index is underlined.

The code for the Display block is shown in Figure 5.9, and illustrates how many output-only blocks work. All the action takes place in the `tick` method, which examines its inputs and sends their values to some I/O device, in this case, the screen via Tk "widgets."

5.3 A MIDI Synthesizer

In this section, I present another application implemented in the SR domain—a MIDI (Musical Instrument Digital Interface) sound synthesizer. This example illustrates many features of the SR domain, including its ability to handle both control and data, incorporate other models of computation, one-to-many communication, and feedback.

The synthesizer decodes a serial MIDI stream and uses it to produce sound using a digital-to-analog converter. All the decoding and control is done by custom SR blocks written in C++; the waveform synthesis is done with existing SDF blocks using an FM (frequency modulation) algorithm.

In this section, I describe the MIDI protocol, the technique of FM synthesis, and how I implemented the synthesizer in Ptolemy.

```
# class SRDisplay
#
# INPUTS leftmost - The index of the leftmost character to display
#        cursor - The index of the cursor position
#        string - The string to display
class SRDisplay {
    inherit SRItclStar itk::Toplevel
    constructor {args} {
        set displayWidth 10
        itk_component add display {
            label $itk_interior.display -width $displayWidth \
               -justify left
        }
        pack $itk_component(display)
        set displayText ""
        set leftmostChar 0
        set cursorPos -1
        eval itk_initialize $args
    }
    method tick {} {
        set string [read string]
        if { $string != "absent" && $string != "unknown" } {
            set displayText $string
            set cursor [read cursor]
            set leftmost [read leftmost]
            if { $cursor != "unknown" && $cursor != "absent" } {
                set cursorPos $cursor
            } else {
                set cursorPos -1
            }
            if { $leftmost != "unknown" && $leftmost != "absent" } {
                set leftmostChar $leftmost
            }
            $itk_component(display) configure -text \
                    [string range "$displayText           " \
                      $leftmostChar \
                      [expr $leftmostChar + $displayWidth]]
            $itk_component(display) configure -underline \
                    [expr $cursorPos - $leftmostChar]
        }
    }
    variable cursorPos
    variable leftmostChar
    variable displayText
    variable displayWidth
}
```

Figure 5.9 Itcl code for the Display block of the Digital Address Book

5.3.1 The MIDI Protocol

The MIDI protocol* was designed to allow a single musician to control multiple synthesizers with a single keyboard. Primarily, it sends note-on and note-off messages that include a note's pitch and how hard it was struck. It is a real-time protocol, so a note-on message causes a note to begin sounding immediately and held until the corresponding note-off message arrives.

MIDI is a unidirectional asynchronous byte-oriented serial protocol that resembles RS-232. In my implementation, I used a keyboard that sends MIDI messages at 38.4 kBaud at RS-232 levels, so it connected directly to the serial port of a Sun workstation.

MIDI defines a series of messages, summarized in Table 5.3. Each begins with a status byte indicating the message type followed by a sequence of data bytes. The most significant bit of each transmitted byte is set for status bytes, cleared for data.

To save bandwidth, MIDI uses something called "running status" where a message's status byte is omitted when the last message had the same status. For example, to send a series of note on commands to Channel 0, a single 90 status byte can be transmitted, followed by pairs of data bytes indicating the pitch and velocity of each note.

My synthesizer responds to Note On, Note Off, Control Change, and Channel Pitch Wheel messages. Control Change messages affect parameters controlling the timbre of the notes. Channel Pitch Wheel messages shift the frequencies of all sounding voices by up to two half-steps in either direction.

5.3.2 FM Sound Synthesis

A natural-sounding musical tone, such as a note struck on a piano, generally consists of a fundamental frequency accompanied by har-

*MIDI is defined by the still-evolving *MIDI 1.0 Detailed Specification* [51], but books such as Rothstein [59] or Roads [57] have more readable descriptions.

	Meaning	Status	Data 1	Data 2
Voices	Note Off	8*c*[1]	Pitch[2]	Velocity[3]
	Note On	9*c*[1]	Pitch[2]	Velocity[3]
	Note Aftertouch	A*c*[1]	Pitch[2]	Value
	Control Change	B*c*[1]	Controller	Value
	Program Change	C*c*[1]	Program	
	Channel Aftertouch	D*c*[1]	Value	
	Channel Pitch Wheel	E*c*[1]	LSB[4]	MSB
System Common	System Exclusive	F0	Mfg. ID[5]	...
	MTC Quarter Frame	F1	Time Code	
	Song Position Pointer[6]	F2	LSB	MSB
	Song Select[6]	F3	Number	
	Tune Request	F6		
	End of System Exclusive	F7		
System Realtime[9]	Timing Clock[7]	F8		
	Start[6]	FA		
	Continue[6]	FB		
	Stop[6]	FC		
	Active Sensing[8]	FE		
	System Reset	FF		

[1] The lower four bits indicates the channel.
[2] The pitch in halfsteps. Middle C is 60.
[3] A velocity of 64 is neural. A velocity of 0 is equivalent to note off.
[4] The first data byte contains the seven least significant bits; the second contains the next seven bits. Hex 2000 is neutral.
[5] The manufacturer ID byte provides a way to interpret the arbitrary number of data bytes that follow. An F7 status terminates the sequence.
[6] Used for sequencer control.
[7] May be sent out at a rate of 24 per quarter note for synchronization.
[8] Sent out every 300 ms if there has been no other activity to indicate the presence of a MIDI connection.
[9] System Realtime messages may "interrupt" any data stream.

Table 5.3 MIDI messages. All numbers are in hexadecimal.

monics at integer multiples. Although the overall loudness of the note generally decays after it is first struck, the relative strengths of the harmonics usually evolve over time in more complex ways.

It is a challenge to synthesize natural-sounding musical tones because of their need for dynamically-changing harmonics. Additive synthesis,* where the waveform is created by summing sinewaves, is an obvious approach, but is both computationally intensive and demands a lot of difficult-to-obtain data. Subtractive synthesis takes a complementary approach by sending an easy-to-generate, harmonically rich waveform (such as a square or sawtooth wave) through filters to produce the final sound. While closely resembling many real-world mechanisms for generating sounds (e.g., brass instruments filter the sound of vibrating lips), interesting sounds require complex time-varying filters.

In 1973, John Chowning introduced the idea of FM synthesis [22] for synthesizing tones. His key observation was that FM waveforms are easy to produce and have the characteristics of natural sound. In particular, their harmonics can be made to fall at integer multiples of the fundamental and the relative amplitudes of these harmonics can be controlled in complex ways by varying a single parameter.

All of these effects can be seen in the FM equation and its sine expansion,

$$\begin{aligned} y(nT) &= \sin(\theta_c + I\sin(\theta_m)) \qquad (5.1)\\ &= J_0(I)\sin(\theta_c) + \\ &\quad J_1(I)(\sin(\theta_c+\theta_m) - \sin(\theta_c-\theta_m)) + \\ &\quad J_2(I)(\sin(\theta_c+2\theta_m) + \sin(\theta_c-2\theta_m)) + \\ &\quad J_3(I)(\sin(\theta_c+3\theta_m) - \sin(\theta_c-3\theta_m)) + \\ &\quad \vdots \end{aligned}$$

*For more information about additive synthesis and computer music in general, see Roads' extensive tutorial [57]. Moore's book [52] has a more detailed description of FM synthesis.

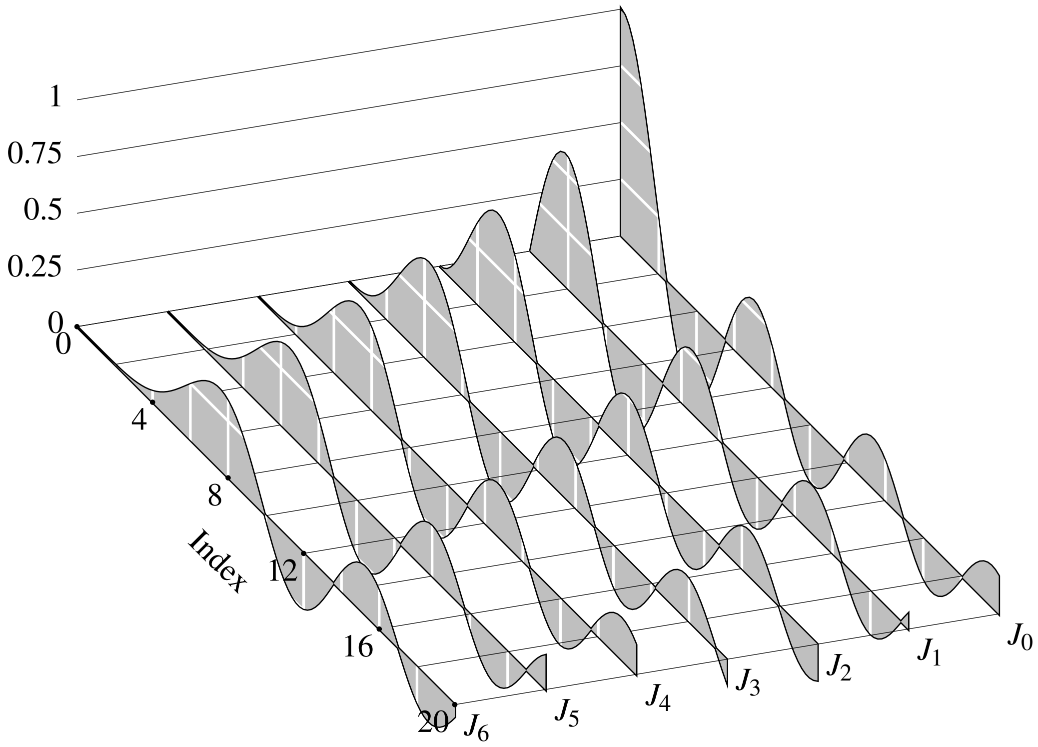


Figure 5.10 The first seven Bessel functions. For a given index of modulation, I, the amplitude of the kth harmonic comes from $J_k(I)$.

where

$$\theta_c = 2\pi f_c nT \qquad \theta_m = 2\pi f_m nT.$$

The carrier frequency, f_c, is the fundamental, and when the modulating frequency f_m equals f_c, the harmonics fall at $f_c, 2f_c, 3f_c$, etc. The modulation index I affects the relative amplitudes of the harmonics through Bessel functions of the first kind. In general, the higher harmonics die out and larger I means more harmonics, but the behavior of lower harmonics is more complex—see Figure 5.10.

Two more things are needed to produce a tone using FM synthesis. The FM equation (5.1) needs to be scaled by a time-varying envelope function, and the index of modulation needs to be controlled for the duration of the note. A typical envelope function starts suddenly when the note is first struck (its attack), then decays to zero when

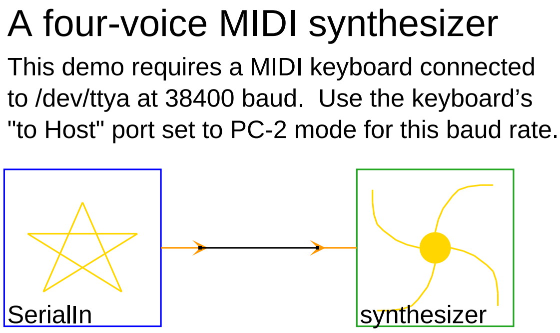


Figure 5.11 The top level of the MIDI synthesizer. The output of the SerialIn block is either the character on the serial port, or absent. The synthesizer block is shown in Figure 5.12.

the note is released (Figure 5.16 shows an example). It is reasonable to make the index of modulation proportional to the envelope, since many natural sounds increase their bandwidth when the note is first struck, then slowly decay into a pure tone (note that when I is zero, the FM equation produces a sinewave).

5.3.3 Implementation of the Synthesizer

I implemented the synthesizer in Ptolemy using both the SR and SDF domains. The top two levels of hierarchy (Figures 5.11 and 5.12), are implemented in the SR domain, and are responsible for decoding the MIDI stream, keeping track of which voices are sounding, and generating the envelopes for the notes. The bottom two levels, Figures 5.17 and 5.18, are implemented in the SDF domain and are responsible for synthesizing the FM waveforms, summing them, and sending them to the speaker. The synthesizer runs in real time with four voices running at 8 kHz on a SPARCStation 10.

At the top level (Figure 5.11), the single input to the system—a serial MIDI stream—enters via the SerialIn block. This block's output emits either the most-recently-sent MIDI byte, or is absent when no

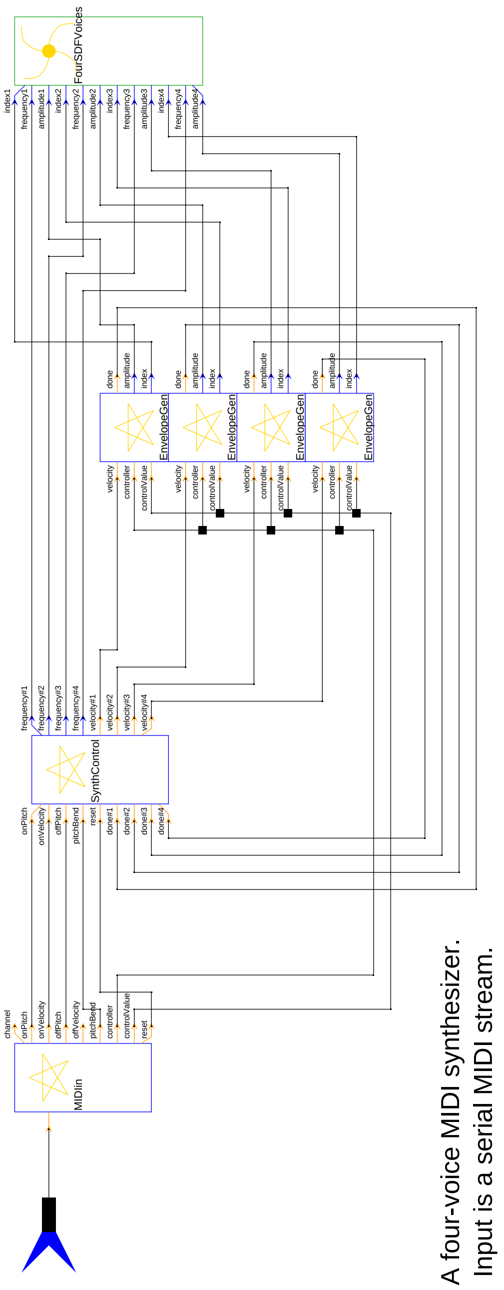


Figure 5.12 The synthesizer block. Starting from the input, the MIDIin block translates the serial MIDI stream into commands for the polyphony controller, the SynthControl block. This sends frequencies to the sound synthesis block and velocities to the four EnvelopeGen blocks.

new byte has arrived.

The MIDIin Block

The MIDIin block decodes a MIDI stream by keeping track of the running status and recent data bytes to produce note on, note off, control change, and reset events. Its primary function is serial-to-parallel conversion; when a complete note-on message arrives, its channel, pitch, and velocity are emitted in a single instant. Note off, pitch bend, and control change messages are handled similarly. I coded this as a strict block in C++, and it is essentially a state machine, albeit one with a significant need to handle data. See Figures 5.13–5.15.

The SynthControl Block

The SynthControl block is responsible for handling polyphony and the effects of the pitch wheel. Its main inputs are *onPitch*, *onVelocity* and *offPitch*, which receive events from the MIDIin block. The block maintains an array of pitches of currently-sounding voices and distributes this information to the various sound synthesis blocks.

The voices are controlled through *frequency*, *velocity*, and *done* ports. The frequencies of sounding voices come from a look-up table and are each multiplied by a constant derived from the *pitchBend* port that can shift them up or down at most two half steps. The velocity of a voice comes directly from the note-on event and is held until the corresponding note-off event arrives, at which time it becomes absent. There is usually some delay between when a note-off event arrives and when the voice actually stops sounding. An event on a *done* channel signals this. These do not take effect until the next instant to avoid an instantanenous feedback problem.

```
defstar {
    name { MIDIin }
    domain { SR }

    input { name { in }            type { int } }

    output { name { channel }      type { int } }
    output { name { onPitch }      type { int } }
    output { name { onVelocity }   type { int } }
    output { name { offPitch }     type { int } }
    output { name { offVelocity }  type { int } }
    output { name { pitchBend }    type { int } }
    output { name { controller }   type { int } }
    output { name { controlValue } type { int } }
    output { name { reset }        type { int } }

    state { name { lastStatus }    type { int } }
    state { name { byteNum }       type { int } }
    state { name { lastByte }      type { int } }

    private { int nextStatus; int nextByteNum; }

    public {
      inline int isRealtime( int i ) const {
        return i >= 0xf8 && i <= 0xff; }
      inline int isStatus( int i ) const { return i & 0x80; }
      inline int isSystem( int i ) const {
        return i >= 0xf0 && i <= 0xff; }

      enum statusBytes {

        NoteOffPitch = 0x80, NoteOffVelocity = 0x81,
        NoteOnPitch = 0x90, NoteOnVelocity = 0x91,
        ControlChangeController = 0xb0, ControlChangeValue = 0xb1,
        ChannelPitchwheelLSB = 0xe0, ChannelPitchwheelMSB = 0xe1,

        SystemReset = 0xff
      };

    }

    begin {
      nextStatus = lastStatus;
      nextByteNum = byteNum;
    }
```

Figure 5.13 C++ code for the MIDIIn block, first part

```
go {
  if ( in.present() ) {
    int inVal = int(in.get());
    if ( isStatus(inVal) ) {
      if ( isRealtime(inVal) ) {
        if  ( inVal == SystemReset ) {
          reset.emit() << int(TRUE);
        }
      } else {
        nextStatus = inVal;
        nextByteNum = 0;
      }
    } else {
      int status = int(lastStatus);
      if ( isSystem(status) ) {
      } else {
        int byte = int(byteNum);

        switch ( status & 0xf0 | byte ) {

        case NoteOffPitch:
        case NoteOnPitch:
        case ControlChangeController:
        case ChannelPitchwheelLSB:
          lastByte = inVal;
          nextByteNum = 1;
          break;

        case NoteOffVelocity:
          offPitch.emit() << int(lastByte);
          offVelocity.emit() << inVal;
          channel.emit() << int(status & 0xf);
          nextByteNum = 0;
          break;

        case NoteOnVelocity:
          channel.emit() << int(status & 0xf);
          if ( inVal == 0 ) {
            offPitch.emit() << int(lastByte);
            offVelocity.emit() << 64;
          } else {
            onPitch.emit() << int(lastByte);
            onVelocity.emit() << inVal;
          }
          nextByteNum = 0;
          break;
```

Figure 5.14 C++ code for the MIDIIn block, second part

```
          case ControlChangeValue:
            controller.emit() << int(lastByte);
            controlValue.emit() << inVal;
            channel.emit() << int(status & 0xf);
            nextByteNum = 0;
            break;

          case ChannelPitchwheelMSB:
            pitchBend.emit() << ((int(inVal) << 7) | lastByte);
            channel.emit() << int(status & 0xf);
            nextByteNum = 0;
            break;
          }
        }
      }
    }

    if ( !channel.known() )          { channel.makeAbsent(); }
    if ( !onPitch.known() )          { onPitch.makeAbsent(); }
    if ( !onVelocity.known() )       { onVelocity.makeAbsent(); }
    if ( !offPitch.known() )         { offPitch.makeAbsent(); }
    if ( !offVelocity.known() )      { offVelocity.makeAbsent(); }
    if ( !pitchBend.known() )        { pitchBend.makeAbsent(); }
    if ( !controller.known() )       { controller.makeAbsent(); }
    if ( !controlValue.known() )     { controlValue.makeAbsent(); }
    if ( !reset.known() )            { reset.makeAbsent(); }
  }

  tick {
    lastStatus = nextStatus;
    byteNum = nextByteNum;
  }

}
```

Figure 5.15 C++ code for the MIDIIn block, last part

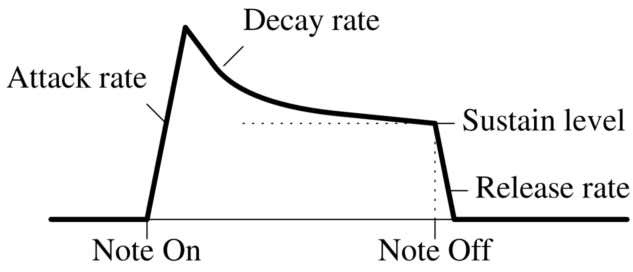


Figure 5.16 An envelope generated by an EnvelopeGen block.

The EnvelopeGen Block

An EnvelopeGen blocks controls both the amplitude and modulation index of each FM synthesis block. The envelopes it generates are controlled by four parameters, shown in Figure 5.16. The attack and release behaviors are both linear; the decay is an exponential fade to the sustain level. These can be set by MIDI Control Change messages that enter through the *controller* and *controlValue* ports, which are connected bus-style to the MIDIin decoder. A knob on the MIDI keyboard can be programmed to generate control change messages for different controllers.

The block behaves as a state machine, looking for velocity events signaling the beginning of a note and the absence of velocity events, signaling the end of a note. The machine has four states, quiet, attack, decay, and release. It changes from quiet to attack and release because of velocity events; it changes from attack to decay and release to quiet based on the envelope state.

The overall amplitude of the envelope is scaled by the velocity of the note. The modulation index is a scaled version of the overall envelope, controlled by a parameter. A larger scaling constant makes the tones sound richer because they have more harmonics.

The FM Synthesis Blocks

Each FM synthesis block (shown in Figure 5.18) has an amplitude, a fundamental frequency, and a modulation index input. The mod-

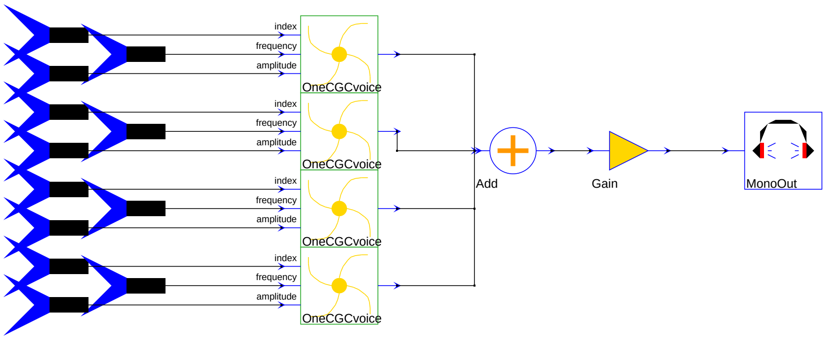


Figure 5.17 The sound synthesis block of the MIDI synthesizer, described using SDF. Four FM synthesis blocks feed their outputs into an adder that sends its output to a speaker.

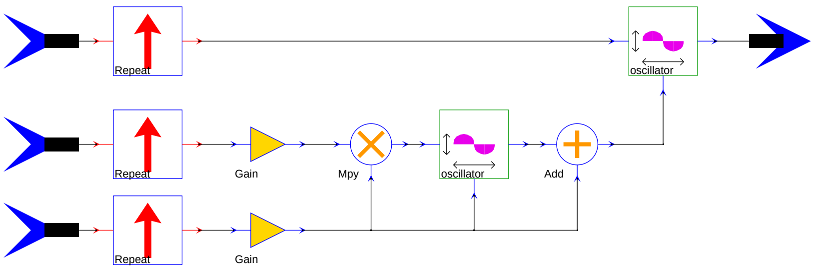


Figure 5.18 An FM synthesis block. The inputs are an amplitude, a center frequency, and an index of modulation. The modulating frequency is the same as the carrier. The repeat blocks (upward arrows) on each input control the number of samples generated per tick of the enclosing SR domain.

ulating frequency is the same as the fundamental, so the harmonics fall at $2f$, $3f$, $4f$, etc.

The FM synthesis blocks are described in Synchronous Dataflow, and take advantage of its ability to describe multi-rate systems. Repeat blocks on each input control how many samples are sent to the digital-to-analog converter per instant. In each instant, exactly one token is placed on each input, so the repeat block copies this as many times as necessary to increase the number of samples per tick. By adjusting this number, the fraction of time the system spends dealing with control versus dataflow can be controlled.

Chapter 6

Conclusions

Films are never completed,
they're only abandoned.
— Anonymous

IN THIS DISSERTATION, I presented a new model of computation for reactive software systems. Called SR, it is the first to combine precise control over when things happen with the ability to assemble systems from heterogeneous pieces. To demonstrate its practicality, I have defined its semantics formally, proven it deterministic, devised an algorithm capable of executing it efficiently and predictably, shown it has a straightforward implementation, and used it to describe some useful, realistic systems.

The formal semantics presented in Chapter 3 showed that SR systems are deterministic and compositional. Introducing an undefined value to the communication channels allows seemingly paradoxical systems to be handled, and requiring the blocks to behave monotonically with respect to this showed these systems are deterministic: they react in exactly one way to any particular input. The semantics also show that any group of SR blocks can be treated as a single block, allowing any part of an SR system to be encapsulated into a library component without loss of expressiveness. This is the primary mechanism for handling complexity through abstraction.

The execution scheme in Chapter 4 determines an order for executing the blocks. The resulting schedule works for all possible inputs, and because all scheduling-related decisions are made before the system is running, there is virtually no run-time scheduling overhead, making execution efficient and predictable. I proved this execution scheme complies with the formal semantics and presented some experimental results that show it is practical for reasonable-sized examples.

Finally, in Chapter 5, I presented a practical implementation of the SR model of computation in Ptolemy, an environment for prototyping heterogeneous systems. In addition to showing the programming interface is fairly straightforward, I presented two examples of real systems specified using SR. One was a digital address book that illustrated SR's suitability for specifying user-interface-dominated systems; the other a MIDI music synthesizer that showed SR's ability to handle control-dominated systems and to assemble heterogeneous subsystems. Both run in real-time thanks to SR's efficient execution algorithm.

6.1 Implications of This Work

Many of the results in this work are very-SR specific (e.g., the theorems in Chapter 4), but some apply more generally. Here are the more far-reaching implications of my work:

Fixed-point semantics are the "right thing" for zero-delay systems with feedback.

Instantaneous feedback loops often cause strange behavior in languages without fixed-point semantics. The VHDL language [45] is typical. To handle zero delay situations, it uses "delta timesteps" that resemble the iterations in my execution scheme, but the number of these steps are unpredictable in general, and may be unbounded, so the simulator can effectively deadlock. Moreover, the simulator has some freedom over the timestep in which a thing occurs, which can lead to nondeterminism.

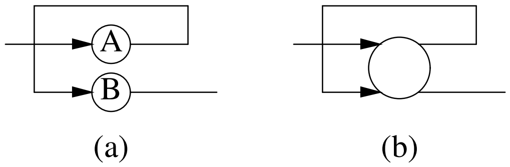


Figure 6.1 An example of non-compositionality in SDF. (a) A correct system. (b) One that deadlocks after blocks A and B have been encapsulated.

As I argued in Chapter 3, fixed-point semantics are both physically realistic and mathematically well-founded, making them an excellent theoretical model of zero-delay systems. Furthermore, Chapter 4 showed that systems having such semantics can be executed efficiently, making them reasonable in practice.

It it my hope that future languages with the need for zero-delay feedback are placed on this firmer theoretical ground.

Partial evaluation solves certain compositionality problems

Many languages impede abstraction by preventing certain kinds of subsystems from being encapsulated, making it harder to build complex systems. A common source of problems is the introduction of unwanted dependencies. Often, encapsulating a subsystem requires its interface to be more synchronized that an unencapsulated version, and this can lead to different behavior, perhaps even deadlock. Lee's Synchronous Dataflow* has this problem, as illustrated in Figure 6.1.

More traditional languages without feedback also have this problem. In the C language, the effect of the AND operator cannot be encapsulated because it does not require all its arguments to be evaluated before it can produce a result. In an expression like

```
i >= 0 && c[i] > 1,
```

the right expression is not evaluated if the left one is false. If this was encapsulated in a C function, both would have to be evaluated every time, which might lead to a memory access violation.

*See Section 2.4.2 on Page 27.

An SR block avoids these problems by being able to execute with only partial information about its inputs. Incomplete inputs generally produce incomplete outputs, but this avoids the needless addition of synchronization that causes these problems.

Chaotic iteration can be a practical way to execute systems

At first glance, chaotic iteration appears too unpredictable to execute practical systems. Whether it converges at all, let alone predictably, is the obvious concern.

My results in Chapter 4 resolve these questions. When chaotic iteration operates on simple, discrete domains with monotonic functions, it is predictable and practical. Using simple dependency information improves the scheme's efficiency, which, on average, seems to be quite a bit better than the theoretical worst case. Of course, executing a system more directly will usually be more efficient, but when this is not possible, such as when a system is assembled heterogeneously, the speed penalty caused by chaotic iteration is not prohibitive.

Recursive strongly-connected component decomposition can produce superior results

Strongly-connected component (SCC) decomposition is a powerful technique for decomposing a graph, but many algorithms stop after applying it once. By contrast, the recursive decomposition scheme I presented in Chapter 4 uses SCC decomposition to ultimately reduce a graph to single nodes. There are many examples where this produces superior results. Bourdoncle's weak topological order* is also a recursive decomposition scheme, but it limits the way an SCC can be further decomposed, diminishing the quality of its results.

*See Page 57

It it my hope that this technique will find application in other problems where a system of equations needs to be solved rapidly. Although it may be less predictable in other domains (e.g., when dealing with real numbers), I expect it will still give better results.

6.2 Future Work

6.2.1 Execution Issues

Synthesizing software from SR system descriptions is an obvious application of this work. The block code generation technique, where code for each block is simply inlined in the order prescribed by the schedule, could be used to synthesize code for a sequential imperative language such as C.

Avoiding the need to explicitly represent "undefined" or "absent" would be an interesting possible optimization. Using more information about the blocks, it might be possible to prove that "undefined" would never appear in acyclic sections of a system. The effect of absent might be possible to reproduce by simply not executing certain reactive blocks.

Since the semantics of SR are not directly tied to a particular execution style, others are possible. Distributed execution is an obvious alternative. A good starting point would be the work of Caspi et al., [19] which executes OC programs (see Section 2.3.3) on distributed processors communicating through FIFO buffers. Their algorithm copies the program, removes redundant sections, and inserts communication that sends variable values to where they are needed and, as a side effect, synchronizes the system. Using the same approach with SR would be easier since there are no decision points in SR schedules.

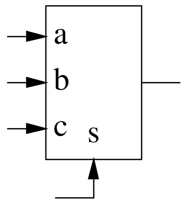
More work can be done with execution time estimation. Execution time of an SR system was designed to be fairly easy to estimate by simply adding the worst-case times for each block according to the schedule, but this may be too pessimistic. The times for multiple invocations of the same block in an instant may differ substantially yet predictably. Even better bounds would probably have to take data dependencies into account.

Making more information about SR blocks available to the scheduler might improve execution speed. Currently, I only have a rather heavy-handed “this input does not matter” flag. It might be better to supply “this input does not affect these outputs” information. Information like “these two outputs are always going to be defined simultaneously” might further speed convergence. In all cases, these merely give the scheduler a better idea about how fast the network will converge without affecting the semantics.

Although grouping a set of SR blocks into a single block does not affect the behavior of a system, it may reduce execution efficiency. The problem is that the execution scheme I devised wants to evaluate blocks an output at a time, yet blocks generally evaluate all their outputs at once, usually because of sharing of intermediate results. For example, when a block containing a subsystem is evaluated, only a few outputs may be needed, yet the schedule for the subsystem will evaluate all of them.

The alternative would be to ask blocks to only evaluate certain outputs to avoid needless computation. One danger is enumerating the exponentially many possible sets of outputs (presumably, there is an optimal schedule for each), but there are probably ways to consider only as many different schedules as there are outputs or blocks in a subsystem.

All of the blocks in an SR system compute monotonic functions, but what is the most efficient way to evaluate these? If they are specified as a C++ or Itcl method, then the most efficient way is to simply execute the method, but if they are specified in some other language, say Esterel or Verilog, then the answer is less obvious. The most efficient way might be to synthesize efficient code using variations of



$$\begin{cases} a & \text{if } s = 0 \\ b & \text{if } s = 1 \\ \sqcap\{a,b,c\} & \text{if } s = \bot \end{cases}$$

Figure 6.2 The universal monotonic multiplexer for the binary CPO $\{\bot, 0, 1\}$. It is monotonic, so a system constructed solely from these will be monotonic. Moreover, any monotonic function can be built with it.

traditional logic synthesis algorithms developed for digital logic.[*] A starting point might be the universal monotonic multiplexer shown in Figure 6.2. With this, it might be possible to apply binary decision diagram (BDD[†]) ideas to manipulate these functions, as BDDs can be thought of as being decompositions of functions into multiplexers.

6.2.2 Language Issues

The biggest open question is how best to describe “native” SR blocks. Strict blocks may be imported from just about any language, and in many cases, this is the best solution, but non-strict blocks are difficult to import. My current solution, using C++ or Itcl, is functional, but not very elegant. I believe some sort of FSM description style would work well, but it would have to be quite a bit more sophisticated than the simple state diagrams of Section 2.3.2. None of the blocks in my MIDI synthesizer could be succinctly described as state diagrams. An alternative would be to use an FSM language like Esterel [7] or even a simplified subset of something like the popular Verilog [69] or VHDL [45] languages. All of these could be given non-strict semantics. The biggest challenge in “solving” this problem is that it is not quantitative—the best solution is the one that people like the most.

[*] De Micheli [24] is a good starting point for this field.

[†] Attributed to Bryant [10], these efficient representations of binary functions are currently the rage in the logic synthesis community.

The semantics of SR are sufficiently abstract to allow for many language variations. One possible variation would be to use non-flat CPOs in the communication channels, although their height would have to remain finite to ensure convergence. This would require minor modifications to the execution scheme, but might ultimately lead to more efficient schedules. Such CPOs could model things that take on a sequence of values in an instant, such as bounded-length FIFOs, program counters, etc., although all of these can be simulated less efficiently with multiple wires. Another application would be to layer clocks on the channels more elegantly.* The first level of the CPOs would contain only clock information; the second would have values. Execution would first establish the presence or absence of all the signals, then establish their values.

SR, as it stands, does not provide facilities for preempting or controlling the execution of subsystems, but this could be added without changing the semantics. The research done on Esterel† has shown that the ability to start, stop, and reset a subsystem is both useful and sufficient. Since subsystems appear as SR blocks, the semantics are clear: the function computed by the block is either the function computed by the subsystem or "all outputs absent" if the system is not being run. The wormhole already has a facility resembling this: if all its outputs are absent, the enclosed subsystem is not executed.

*Gérard Berry pointed this out to me.
†See Section 2.3.5 on Page 21.

Bibliography

The pages on which each citation appears are listed in parentheses on the right. The text set in `Courier` is a world-wide web address.

[1] Lloyd Allison. (34, 38)
A Practical Introduction to Denotational Semantics.
Cambridge University Press, 1986.

[2] H. Bekić. (67)
Definable operations in general algebras, and the theory of automata and flowcharts.
In C. B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer-Verlag, 1984.

[3] Albert Benveniste and Gérard Berry. (14)
The synchronous approach to reactive real-time systems.
Proceedings of the IEEE, 79(9):1270–1282, September 1991.

[4] G[érard] Berry. (21)
The constructive semantics of pure Esterel.
Book in preparation, 1996.
`ftp://cma.cma.fr/esterel/constructiveness.ps.gz`

[5] G[érard] Berry and L. Cosserat. (4)
The ESTEREL synchronous programming language and its mathematical semantics.
In S. D. Brooks, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer-Verlag, 1984.

[6] Gérard Berry and Alain Girault. (22)
Circuit generation for verification of Esterel programs.
Conference paper in preparation.

[7] Gérard Berry and Georges Gonthier. (21, 137)
The Esterel synchronous programming language: Design, semantics, implementation.
Science of Computer Programming, 19(2):87–152, November 1992.
`ftp://cma.cma.fr/esterel/BerryGonthierSCP.ps.Z`

[8] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A[shford] Lee. (28)
Software Synthesis from Dataflow Graphs.
Kluwer, 1996.

[9] François Bourdoncle. (57)
Efficient chaotic iteration strategies with widenings.
In *Formal Methods in Programming and Their Applications: International Conference Proceedings*, volume 735 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, June 1993. Springer-Verlag.

http://www.ensmp.fr/~bourdonc/fmpa93.ps.Z

[10] Randal E. Bryant. (136)
Graph-based algorithms for boolean function manipulation.
IEEE Transactions on Computers, C-35(8):677–691, August 1986.

[11] Randal E. Bryant. (36)
Algorithmic aspects of symbolic switch network analysis.
IEEE Transactions on Computer-Aided Design, CAD-6(4):618–633, July 1987.

[12] Randal E. Bryant. (36)
Boolean analysis of MOS circuits.
IEEE Transactions on Computer-Aided Design, CAD-6(4):634–649, July 1987.

[13] Randal E. Bryant. (36)
A switch-level model and simulator for MOS digital systems.
IEEE Transactions on Computers, C-33(2):160–177, February 1987.

[14] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. (36)
COSMOS: A compiled simulator for MOS circuits.
In *Proceedings of the 24th Design Automation Conference*, pages 9–16, 1987.

[15] Janusz A. Brzozowski and Carl-Johan H. Seger. (36)
Asynchronous Circuits.
Springer-Verlag, 1995.

[16] J[oseph] [Tobin] Buck, S[oonhoi] Ha, E[dward] A[shford] Lee, and D[avid] G. Messerschmitt. (101, 101)
Ptolemy: A mixed-paradigm simulation/prototyping platform in C++.
In *Proceedings of the C++ At Work Conference*, Santa Clara, CA, November 1991.
http://ptolemy.berkeley.edu/

[17] Joseph [Tobin] Buck, Soonhoi Ha, Edward A[shford] Lee, and David G. Messerschmitt. (101, 101)
Ptolemy: A framework for simulating and prototyping heterogeneous systems.
International Journal of Computer Simulation, 4:155–182, April 1994.
http://ptolemy.eecs.berkeley.edu/papers/JEurSim/index.html

[18] W. F. Buhl, A. E. Erdem, F. C. Winkelmann, and E. F. Sowell. (57)
Recent improvements in SPARK: Strong component decomposition, multivalued objects, and graphical interface.
Technical Report LBL-33906, Lawrence Berkeley Laboratory, August 1993.

[19] Paul Caspi, Alain Girault, and Daniel Pilaud. (17, 135)
Distributing reactive systems.
In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, October 1994. ISCA.
http://ptolemy.eecs.berkeley.edu/~girault/Publications/Pdcs94/

[20] P[aul] Caspi, D[aniel] Pilaud, N[icholas] Halbwachs, and J. Plaice. (23)
LUSTRE: A declarative language for programming synchronous systems.
In *ACM Symposium on Principles of Programming Languages*, Munich, January 1987. ACM.

[21] D. Chazan and W. Miranker. (56)
Chaotic relaxation.
Linear Algebra and Its Applications, 2(2):199–222, 1969.

[22] John M. Chowning. (121)
The synthesis of complex audio spectra by means of frequency modulation.
Journal of the Audio Engineering Society, 21(7):526–534, September 1973.

[23] B. A. Davey and H. A. Priestley. (38, 47, 59)
Introduction to Lattices and Order.
Cambridge University Press, 1990.

[24] G. De Micheli. (136)
Synthesis and Optimization of Digital Circuits.
McGraw-Hill, 1994.

[25] Stephen Edwards. (57)
An Esterel compiler for a synchronous/reactive development system.
Master's thesis, University of California, Berkeley, June 1994.
Available as UCB/ERL M94/43.

[26] E. B. Eichelberger. (36)
Hazard detection in combinational and sequential switching circuits.
IBM Journal of Research and Development, 9(2):90–99, March 1965.

[27] András Frank. (85)
How to make a digraph strongly connected.
Combinatorica, 1(2):145–153, 1981.

[28] Michael R. Garey and David S. Johnson. (86)
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W. H. Freeman and Company, 1979.

[29] Carl A. Gunter. (34, 38)
Semantics of Programming Languages.
MIT Press, 1992.

[30] N[icholas] Halbwachs. (18)
Synchronous Programming of Reactive Systems.
Kluwer, 1993.

[31] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. (23)
The synchronous data flow programming language LUSTRE.
Proceedings of the IEEE, 79(9):1305–1320, September 1991.
`ftp://ftp.imag.fr/pub/SPECTRE/LUSTRE/PAPERS/lustre.ieee.ps.gz`

[32] Nicholas Halbwachs, Fabienne Lagnier, and Christophe Ratel. (14)
Programming and verifying real-time systems by means of the synchronous data-flow languages LUSTRE.
IEEE Transactions on Software Engineering, 18(9):786–793, September 1992.
`ftp://ftp.imag.fr/pub/SPECTRE/LUSTRE/PAPERS/lustre.tse.ps.gz`

[33] D. Harel. (18)
Statecharts: A visual formalism for complex systems.
Science of Computer Programming, 8(3):231–274, June 1987.

[34] D. Harel and A. Pnueli. (1)
On the Development of Reactive Systems, volume 13 of *NATO ASI Series. Series F, Computer and Systems Sciences*, pages 477–498.
Springer-Verlag, 1985.

[35] J. Hopcroft and J. Ullman. (11)
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, 1979.

[36] C. Huizing, R. Gerth, and W. P. de Roever. (18)
Modelling Statecharts behavior in a fully abstract way.
In *CAAP '88: 13th Colloquium on Trees in Algebra and Programming*, volume 299 of *Lecture Notes in Computer Science*, pages 271–294, Nancy, France, March 1988. Springer-Verlag.

[37] Larry G. Jones. (57)
Efficient evaluation of circular attribute grammars.
ACM Transactions on Programming Languages and Systems, 12(3):429–462, July 1990.

[38] Gilles Kahn. (26, 34)
The semantics of a simple language for parallel programming.
In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[39] Gilles Kahn and David B. MacQueen. (26)
Coroutines and networks of parallel processes.
In *Information Processing 77: Proceedings of IFIP Congress 77*, pages 993–998, Toronto, Canada, August 1977. North-Holland.

[40] Samit Kuller, October 1996. (86)
Personal communication.
`http://www.cs.umd.edu/~samir/`

[41] Edward Ashford Lee. (28)
A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors.
PhD thesis, University of California, Berkeley, 1986.
Available as UCB/ERL M86/54.

[42] Edward Ashford Lee and David G. Messerschmitt. (27)
Static scheduling of synchronous data flow programs for digital signal processing.
IEEE Transactions on Computers, C-36(1):24–35, January 1987.

[43] Edward A[shford] Lee and David G. Messerschmitt. (27)
Synchronous data flow.
Proceedings of the IEEE, 75(9):1235–1245, September 1987.

[44] Edward A[shford] Lee and Thomas M. Parks. (27)
Dataflow process networks.
Proceedings of the IEEE, 83(5):773–801, May 1995.
`http://ptolemy.eecs.berkeley.edu/papers/processNets`

[45] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. (5, 56, 132, 137)
VHDL: Hardware Description and Design.
Kluwer, 1989.

[46] Sharad Malik. (37)
Analysis of cyclic combinational circuits.
In *1993 IEEE/ACM International Conference on Computer-Aided Design: Digest of Technical Papers*, pages 618–625, Santa Clara, CA, November 1993. IEEE Computer Society Press.

[47] Sharad Malik. (37)
Analysis of cyclic combinational circuits.
IEEE Transactions on Computer-Aided Design, 13(7):950–956, July 1994.

[48] F. Maraninchi. (18)
The Argos language: Graphical representation of automata and description of reactive systems.
In *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.
`ftp://ftp.imag.fr/pub/SPECTRE/ARGONAUTE/ArgosIEEEVisual.ps.gz`

[49] F. Maraninchi. (18)
Operational and compositional semantics of synchronous automaton compositions.
In *CONCUR '92. Third International Conference on Concurrency Theory.*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564, Stony Brook, NY, August 1992. Springer-Verlag.
`ftp://ftp.imag.fr/pub/SPECTRE/ARGONAUTE/README.html`

[50] Michael J. McLennan. (108)
`[incr Tcl]`, 1993.
`http://www.tcltk.com/itcl/`

[51] MIDI Manufacturers Association, Los Angeles, California. (117)
MIDI 1.0 Detailed Specification, 1984–present.
`http://home.earthlink.net/~mma/`

[52] F. Richard Moore. (121)
Elements of Computer Music.
Prentice Hall, 1990.

[53] J. K. Ousterhout. (101, 108)
Tcl and the Tk Toolkit.
Addison-Wesley, 1994.

[54] Thomas M. Parks. (26)
Bounded Scheduling of Process Networks.
PhD thesis, University of California, Berkeley, 1995.
Available as UCB/ERL M95/105.
`http://ptolemy.eecs.berkeley.edu/`

[55] Donald O. Pederson. (35)
A historical review of circuit simulation.
IEEE Transactions on Circuits and Systems, CAS-31(1):103–111, January 1984.

[56] A. Pnueli and M. Shalev. (18)
What is in a step: On the semantics of Statecharts.
In *TACS'91: Theoretical Aspects of Computer Software: International Conference*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264, Sendai, Japan, September 1991. Springer-Verlag.

[57] Curtis Roads. (117, 121)
The Computer Music Tutorial.
MIT Press, 1996.

[58] François Robert. (56, 66)
Discrete Iterations: A Metric Study, volume 6 of *Springer Series in Computational Mathematics*.

Springer-Verlag, 1986.

[59] Joseph Rothstein. (117)
MIDI: A Comprehensive Introduction.
A-R Editions, Madison, Wisconsin, 1992.

[60] David A. Schmidt. (34)
Denotational Semantics: A Methodology for Language Development.
Allyn and Bacon, 1986.

[61] Dana Scott. (32)
Lattice theory, data types and semantics.
In Randall Rustin, editor, *Formal Semantics of Programming Languages*, pages 65–106. Prentice Hall, 1972.

[62] Dana Scott and Christopher Strachey. (32)
Toward a mathematical semantics for computer languages.
In *Proceedings of the Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, April 1971.

[63] C[arl]-J[ohan] [H.] Seger and J[anusz] A. Brzozowski. (36)
Generalized ternary simulation of sequential circuits.
Informatique théorique et Applications, 28(3–4):159–186, 1994.

[64] Thomas R. Shiple, Gérard Berry, and Hervé Touati. (23, 38)
Constructive analysis of cyclic circuits.
In *Proceedings of the European Design and Test Conference*, March 1996.
`ftp://ic.eecs.berkeley.edu/pub/Memos_Conference/edtc96.SBT.ps.Z`

[65] Joseph Stoy. (32, 34)
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
MIT Press, 1977.

[66] Bjarne Stroustrup. (101)
The C++ Programming Language.
Addison-Wesley, second edition, 1991.

[67] Robert Tarjan. (86)
Depth-first search and linear graph algorithms.
SIAM Journal on Computing, 1(2):146–160, June 1972.

[68] The C2A Group. (17)
Projet synchrone: Les formats communs des languages synchrones (common formats for the synchronous languages).
Technical Report 157, INRIA, June 1993.
Translated from the French by Wendell Baker.
`http://www.baker.com/wbaker/publications/index.html`

[69] Donald E. Thomas and Philip R. Moorby. (56, 137)
The Verilog Hardware Description Language.
Kluwer, 1991.

[70] Franklyn Turbak, David Gifford, and Brian Reistad. (38)
Applied semantics of programming languages.
Unpublished textbook, 1995.

[71] Jan van Leeuwen, editor. (38)
Handbook of Theoretical Computer Science.
Elsevier/MIT Press, 1990.

[72] Michael von der Beeck. (18)
A comparison of Statecharts variants.
In *Formal Techniques in Real-Time and Fault-Tolerant Systems: Third International Symposium Proceedings*, volume 863 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[73] G. Winskel. (34, 38, 67, 75)
The Formal Semantics of Programming Languages: An Introduction.
Foundations of Computing. MIT Press, 1993.

Index

Colophon

I typeset this in Times Roman and Courier using $\LaTeX\ 2_\varepsilon$ on a Sun workstation. The figures were done using gpic. The graphs were done by pstricks macros generated by a Perl script. The block diagrams in Chapter 5 are Postscript generated from Ptolemy.

The design is a result of the tension between the university's layout rules for dissertations and my desire to make it readable. Their most draconian rule is that the body must be double-spaced, which would look fine if I were using a typewriter.

Most people, when first seeing the design of this dissertation, ask why the columns are so thin. My choice of column width was based on the typographically ideal line width of sixty-six characters, a number based on hundreds of years of typographic experience. This appears narrow on letter-sized paper for twelve point Times Roman because letter-sized paper, again, was designed for typewriters. Well-typeset books are either physically smaller or use multiple columns.

Typography is actually quite a bit like engineering. When done properly, it should go unnoticed.