A Modal Arithmetic for Reasoning About Multi-Level Systems of Finite State Machines

Victor J. Yodaiken

Computer and Information Science Department
University of Massachusetts

COINS Technical Report 91-17
September 1990

A MODAL ARITHMETIC FOR REASONING ABOUT MULTI-LEVEL SYSTEMS OF FINITE STATE MACHINES

A Dissertation Presented

by

VICTOR . J. YODAIKEN [1]

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1990

Department of Computer and Information Science

[1]This research was funded in part by the Office of Naval Research under contract N00014-85-K-0398.

ACKNOWLEDGEMENTS

Krithi Ramamritham was an ideal adviser. He suggested that I look at temporal logics, kept me focused on the problem, and was unfailingly encouraging. Ernie Manes went above and beyond the call of duty in decoding some particularly obscure versions of this work. Dave Barrington provided insight into automata theory and was willing to repeat explanations until they made sense. Jack Stankovik's apparent willingness to believe that something practical might eventually emerge from the project was also very helpful.

Marc Baumslag helped me clarify my thinking and exposition, found the Gecseg book, and was a good friend through this long process. Subhashish Mazumdar's witty conversational style and assistance with the intricacies of modal logic are greatly appreciated. Badrinath, Ugo Buy, Panos Chrysanthos, Chia Shen, Al Hough, and Duane Bailey made up for the lack of windows in our office. Renee Kumar saved me from the bureaucracy more than a few times.

Finally, thanks to Beth Kirkhart for her assistance in life and desert navigation.

To my grandparents, especially K.B. .

ABSTRACT

A MODAL ARITHMETIC FOR REASONING ABOUT MULTI-LEVEL SYSTEMS OF FINITE STATE MACHINES

September 1990

VICTOR YODAIKEN
B.A. COLUMBIA UNIVERSITY
M.S. UNIVERSITY OF SOUTH CAROLINA
Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Krithi Ramamritham

This dissertation advances a novel approach to formal specification and verification of systems-level computation. The approach is based on a purely finite state model of computation, and makes use of algebraic and syntactic techniques which have not been previously applied to the problem. The approach makes use of a modal extension of the primitive recursive functions to specify system behavior, and uses an algebraic *feedback product* of automata to provide semantic content for specifications. The modal functions are shown to provide highly compact and expressive notation for defining, composing, and verifying the properties of large-scale finite state machines. The feedback product allows for a very general model of composition, multi-level dynamic behavior, and concurrency. Techniques are developed for specifying both detailed operation of algorithms, and abstract system properties such as liveness and safety. Several significant examples are provided to illustrate application of the method to complex algorithms and designs.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v
ABSTRACT vi
LIST OF FIGURES ix
LIST OF DEFINITIONS xi

CHAPTER

1. INTRODUCTION 1
   1.1 The problem: Mathematical models of system-level computation. . . 1
   1.2 Contributions 2
   1.3 The modal arithmetic. 3
   1.4 Outline 5
   1.5 Related literature 7
      1.5.1 Sources and related formal methods 7
      1.5.2 Comparison with temporal logic 9
   1.6 Summary 11

2. SYNTAX AND SEMANTICS 12
   2.1 The modal arithmetic 13
      2.1.1 The initial functions 15
      2.1.2 The composed functions 18
      2.1.3 Applications of *after* 19
   2.2 Transducers and traces 20
      2.2.1 Flat transducers 20
      2.2.2 Product form transducers 21
      2.2.3 Transition precedence. 23
   2.3 Review of the Primitive Recursive Functions 25
   2.4 A formal definition of the modal p.r. functions. 27
   2.5 Interpretation 28
   2.6 Summary 31

3. SPECIFICATION AND VERIFICATION 32
   3.1 Specification Style 34
      3.1.1 Modal recursion 34
      3.1.2 Modal grammars: Specifications of systems 36
      3.1.3 An example. 37
   3.2 Proofs 40
      3.2.1 Inherited algebra 40
      3.2.2 Modal proofs 41
      3.2.3 Path division and path induction 42
      3.2.4 Distributive m.p.r. theorems 43
      3.2.5 Reasoning about components 45
      3.2.6 Reasoning about *precedes* 46
   3.3 Exact modal grammars 46
      3.3.1 Syntax of exact grammars 47

3.3.2 The exact grammar theorem 48
3.4 Summary 51

4. TEMPORAL LOGIC STYLE FUNCTIONALS 53
4.1 The interval functions 54
4.2 Branching time 56
4.2.1 The operators of branching time 56
4.2.2 The m.p.r. branching time functionals 57
4.3 Proofs with branching functionals 58
4.3.1 Verification of the analogy 59
4.3.2 Proof of the UB axioms 60
4.3.3 Until 62
4.4 Pumping theorems 64
4.4.1 A congruence based on configurations 64
4.4.2 A congruence based on relative transition ordering 65
4.4.3 The intersection of two congruences 68
4.5 Summary 71

5. REAL TIME 72
5.1 Real time techniques 73
5.2 A real-time priority queue 75
5.2.1 The specification 76
5.2.2 Implementation 78
5.2.3 The timer 79
5.2.4 The queue module 79
5.2.5 Putting it together 81
5.2.6 Verification. 83
5.3 A fragment of Futurebus+ arbitration 86
5.4 Summary 91

6. A FAULT TOLERANT MESSAGE PROTOCOL 92
6.1 The algorithm 95
6.2 Summary 101

7. CONCLUSION 103
7.1 Dissertation Summary 103
7.2 Future Work 104
7.2.1 Applications of M.p.r. arithmetic 104
7.2.2 Theoretical investigations 106

BIBLIOGRAPHY 108

LIST OF FIGURES

1.1 Multi-level temporal logic style assertions 6

2.1 Sequence functions 13

2.2 A product form transducer. 14

2.3 Informal definition of the modal functionals. 19

3.1 A fifo queue 35

3.2 A lifo queue 36

3.3 A modal grammar 36

3.4 A clocked storage cell 38

3.5 A clocked memory bank 38

3.6 A clocked fifo queue 39

3.7 Goodstein's rules for primitive recursive arithmetic 41

5.1 A complete high level specification of the real-time queue 78

5.2 The timer specification. 79

5.3 The queue module. 80

5.4 Specification of the implementation. 81

5.5 Pin logic levels 87

5.6 Signal stability 87

5.7 The arbitration bus 87

5.8 The interface of a bus device 88

5.9 Futurebus+ system sketch 89

5.10 Competition synchronization on the FutureBus+ 89

5.11 Device competition algorithm for Futurebus+ 90

5.12 Safety condition for Futurebus+ 90
5.13 Liveness condition for Futurebus+ 91
6.1 Messages on the broadcast network 94

LIST OF DEFINITIONS

2.1 $Past(x, y)$ 18

2.2 *Initial* 18

2.3 Extension of enable and f_effect 19

2.4 Informal version of $\Box$ 20

2.5 The flat product form transducers 21

2.6 Δ for flat transducers 21

2.7 The class of product form transducers 22

2.8 The product state set $PS(P)$ 22

2.9 Reflexive, transitive closure of ϕ 23

2.10 $pfactor$ and Δ for product form transducers 23

2.11 The trace language $\mathcal{L}$(P) 23

2.12 *Place* 24

2.13 *RelOrder* 24

2.14 A congruence on *RelOrder* 24

2.15 The congruence classes induced by *RelOrder* 24

2.16 The primitive recursive functions 25

2.17 The class of m.p.r. functions 28

2.18 The value of an m.p.r. function 29

2.19 The evaluation functional, ρ 30

3.1 Satisfaction of a term by a transducer. 33

3.2 Modal recursion. 35

3.3 The class mpr(precedes). 47

3.4 The class $mpr(\text{precedes}, \text{in}_o)$. 47
3.5 Exact grammars. 47
3.6 Black-box equivalence. 48
3.7 The congruence $\approx$ 49
4.1 The interval operators sometime, always, and cumu 54
4.2 The function *Upto*. 54
4.3 Next state functionals. 57
4.4 Henceforth, and Possibly. 57
4.5 The function *Epaths*. 57
4.6 Eventually and indefinitely. 58
4.7 The functional until 58
4.8 The function *Nontriv*. 64
4.9 The configuration congruence $\sim$. 64
4.10 The depth congruence $\approx$. 65
4.11 The depth bound functional *D*. 67
4.12 The depth bound functional *plen*. 69
4.13 The path functionals. 70
5.1 The function *Tcount*. 73
5.2 The function *duration*. 73
5.3 Timed versions of henceforth and eventually. 74

CHAPTER 1

INTRODUCTION

1.1 The problem: Mathematical models of system-level computation.

Formal mathematical specification and verification of computer operating systems, device architectures, or network protocols is hardly ever attempted in current engineering practice. The only exceptions to this rule are systems which have been especially designed (simplified) to facilitate verification and systems that are so safety-critical as to justify the extraordinary effort verification appears to require [13, 41]. In order to confirm the internal design consistency and correct implementation of systems that do not meet these these criteria, engineers rely on robust design techniques, the intuition of skilled practitioners, simulation, and testing. Consequently, errors are often found at an inconveniently late stage in the design process, optimization is limited by the difficulty of discerning which features of the design are truly essential, and intuition is lost in a maze of implementation details. On the other hand, *system-level* computation, which includes operating systems, device and bus architectures, controllers, and computer networks, seems to be exceptionally resistant to formal analysis.

Obstacles to formal analysis. Even relatively simple system-level devices and programs are associated enormous state sets that would be difficult to describe directly. Unfortunately, the nature of system-level computation complicates efforts to subdivide problems into more manageable parts or to abstract away lower level details. Because timing and resource constraints are critical for the correct functioning of many system-level mechanisms, we cannot simply brush aside these issues as implementation details (c.f. [27]). Because there is no single method for connecting concurrent components at the system-level, and because interaction between components is not limited to synchronous message exchanges or some other communication "primitive", we cannot easily view complex systems as networks of communicating *processes* (c.f. [45]). To make these issues more concrete, consider the example of a

bus architecture and arbitration algorithm. Clearly, the real-time required for signal propagation, the order in which modules on the bus assert signals, and the number of modules on the bus may all be critical to design correctness. Similarly, attempting to represent this architecture in terms of processes exchanging messages over synchronous channels would obscure the actual system design and add a layer of complex indirection to what is already a complex and difficult problem.

1.2 Contributions

The *modal primitive recursive (m.p.r.) arithmetic* is proposed here as a solution to the problem of formal analysis of system-level computation. The arithmetic includes both a novel formal semantics and a novel formal language. The formal semantics is based on a sophisticated algebraic product of finite automata called the *feedback product.* We believe that this product provides a semantic basis for reasoning about concurrent and composed systems that is richer in structure and more expressive than the standard methods (*c.f.* [15, 2]). The formal language is based on a modal (state dependent) extension of the primitive recursive functions. The functions provide an intuitive and clear means of describing both the detailed workings of complex algorithms and the high level abstract properties of systems. Especially for distributed algorithms and multi-level systems, we believe that the language of m.p.r. functions has significant advantages over previously proposed computational logics and descriptive languages.

In brief, we claim the following.

- A formal language for defining state machines which:
  - Allows for intuitive and highly compact (super-exponential) definitions of automata, without requiring enumeration of states or loss of detail;
  - Is flexible enough to allow for both detailed specification of algorithms and designs at the transistor and pico-second level, and abstract specification of high level properties such as liveness, fairness, and safety.
  - Facilitates clear expression of multi-level properties, e.g., the expression *within board* b *within module* m *within circuit* c *within gate* g Out = 1 .
  - Permits specifications of composite systems to be constructed in a simple and clear manner from specifications of the components.
  - Uses an underlying notion of state function which exposes the causality of state change, and which greatly simplifies reasoning about the dynamic behavior of functions.

- A sophisticated model of concurrency and composition which:
  - Provides a precise, intuitive semantics for encapsulation;
  - Makes a clear distinction between non-determinism and composition;
  - Requires no assumptions about the communication methods used between modules, or the type of concurrency present in the system;
  - Supports a finitary and constructive notion of fairness and liveness.
- A formal method which is integrated into mathematics, so that the theorems and proof methods of integer mathematics are immediately available for analysis of specifications.

1.3 The modal arithmetic.

Implicit specification. During the design or development of a digital system it seems natural to think about state indirectly. For example, one of the correctness conditions of a reliable message transfer might be written:

$$\mathsf{Commit}(\mathsf{source}, \mathsf{m}) \rightarrow \mathsf{Received}(\mathsf{dest}, \mathsf{m}).$$

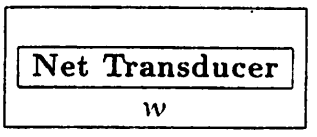
This expression implicitly references the current state of the entire computer network. Pictorially, the expression might be re-written:

$$\boxed{\mathbf{NetState}} \quad \mathsf{Commit}(\mathsf{source}, \mathsf{m}) \rightarrow \mathsf{Received}(\mathsf{dest}, \mathsf{m})$$

The boxed symbol on the left of the expression is a *context*, and represents the current state of the network. The implication on the right is a *formal expression*, with a concrete meaning that depends on the context. The technique of separating semantics (context) from formal expressions is basic to formal logic. The advantage of this approach is that formal expressions can be made independent of the particular details of the context. For example, the boolean function Commit(source, m), can be defined without worrying about how the network is connected, or how many sites are on the net. If Commit had to be written with the state as a parameter, these details would have to be made explicit.

Contexts. In m.p.r. arithmetic, a context consists of a pair (P, w) where P is a finite state transducer (automaton with output) and w is sequence of transition symbols. We call w a *trace* of P and let it represent the sequence of events which have driven

P from its initial configuration. Generally, we restrict our attention to *enabled traces* — traces which do not drive the transducer into an undefined state. A context (P, w) represents the discrete device modeled by P and the current state of that device as determined by w. Redrawing our pictorial example, we get:



$\mathrm{Commit}(\mathrm{source}, \mathrm{m}) \rightarrow \mathrm{Received}(\mathrm{dest}, \mathrm{m})$.

Contexts of composite systems. Classical finite state transducers represent the operation of mechanisms (physical devices or programs) with discrete state functions. Transducers in *feedback product form* represent discrete *systems* constructed from interconnected components. The feedback product[1] is a technique for constructing complex transducers from previously defined transducers. In addition to a state set, a transition function, and an output function, a product form transducer is associated with a list of transducers (the factor transducers), and a list of *feedback functions* which control the activity of the factors. Whenever a product form transducer accepts an input symbol, the feedback functions induce sequences of input for each factor. The induced sequence depends on the current global state, the input symbol, and the *feedback* — the outputs of the factors. Returning to the example of a computer network, a transducer representing such a network might contain several transducers representing individual sites and a transducer representing the communications medium. A single transfer_out transition of the product transducer might correspond to parallel receive transitions in one or more of the factors representing sites.

Formal expressions. M.p.r. expressions are constructed from functions belonging to a modal extension of the primitive recursive (p.r.) functions. The p.r. functions consist of the initial p.r. functions (zero, plus 1, and projection), plus any functions that can be constructed from other p.r. functions using substitution ($f(x) \stackrel{\mathrm{def}}{=} h(g(x))$) and primitive recursion ($f(0, x) \stackrel{\mathrm{def}}{=} g(x), f(r + 1, x) = h(r, x, f(r, x))$). Peter [48] has shown that the class PR of primitive recursive functions includes a great deal of arithmetic and algebra: from multiplication and exponentiation to finite sets and sequences. Since finite state machines a definable with p.r. functions, it would be possible to describe all interesting properties of contexts within the primitive recursive arithmetic. For example, we could have a boolean function $\mathrm{IsEnabled}(P, w, u)$ which would be true if and only if wu was an enabled trace, i.e., iff u was enabled in the

[1]The feedback product used here is derived from the general product described by Gecseg [17].

context (P, w). Our goal, however, is to avoid explicit analysis of contexts in favor of a more abstract approach. To this end, we make use of a technique of the modal logics [32], to make contexts implicit. We define a class MPR of modal functions to include the p.r. functions plus new functions that implicitly reference a context. We define a map $\rho : MPR \rightarrow PR$ which makes context dependencies explicit. The *value* of a m.p.r. expression $f(x)$ in the context of (P, w) is defined to be $(\rho f)(P, w, x)$.

$$\left[\begin{array}{c} \boxed{\begin{array}{c} P \\ w \end{array}} \; f(x) \end{array}\right] = (\rho f)(P, w, x)$$

Modal and arithmetic functions. If f is a p.r. function, e.g., $f(x, y) \stackrel{\text{def}}{=} \lfloor x/y \rfloor$, then f does not depend on the context at all, and $(\rho f)(P, w, x) = f(x)$. But, MPR also contains modal functions that do depend on the context. For example, there is a m.p.r. function *enable* so that *enable*(u) is true iff u is an enabled path in the current context. We define $(\rho\, enable) = IsEnabled$. Thus, *enable*$(u)$ is true in the context of (P, w) iff $IsEnabled(P, w, u)$ is true. There are also m.p.r. functions which refer to future states, to the relative order of previous state transitions, and to the operation and states of the (possibly concurrent) component sub-systems.

1.4 Outline

In chapter 2 we define a formal semantics and syntax for the m.p.r. functions in such a way as to provide precise mathematical meaning for an expressive and intuitive specification language. In chapter 3 we show that every m.p.r. expression specifies some family of transducers and investigate several styles of specification. We can show that every m.p.r. function constructed under a relatively simple syntactic restriction specifies exactly one (minimal) transducer. We show that there is an algorithm for constructing the specified transducer from the m.p.r. function. We show, further, that every transducer can be specified in this way. Chapter 3 also develops proof methods that allow for deduction of properties of transducers and their traces completely within the framework of the m.p.r functions (without construction of either transducers or traces).

Chapter 4 derives m.p.r. functionals which are analogous to the modal modifiers of the branching time logic [5, 16] and the interval temporal logics [42]. We prove that the m.p.r. analogs are faithful to the originals by proving that the axioms of the branching time logic are theorems of the m.p.r. arithmetic. We also show that

the m.p.r. temporal functionals can be given constructive definitions based on the pumping lemma of regular languages [29]. This is somewhat surprising because the temporal logic modifiers are usually defined in terms of unbounded quantification and given infinitary interpretations [50, 3]. We also demonstrate that the temporal functionals have a well-defined and intuitive meaning in multi-level settings. Consider the three assertions in figure1.1. These assertion make use of the temporal functional $\Diamond$ (inevitably) and the m.p.r. functional in (within a component).

1. $(\text{in server.s})(\text{in req\_q})(\text{Element}(m,c) \rightarrow \Diamond \text{Head}(m,c))$
2. $(\text{in server.s})(\text{in req\_q})(\text{Head}(m,c) \rightarrow \Diamond(\text{in resp\_q})\text{Element}(m',s))$
3. $(\text{in server.s})\ (\text{in req\_q})\text{Element}(m,c)$
   $\not\rightarrow \Diamond(\text{in server.s})(\text{in resp\_q})\text{Element}(m',s)$

Figure: 1.1 Multi-level temporal logic style assertions

A m.p.r expression of the form (in C)E refers the the value of the expression E in the context of the component named C. Thus, formal expression of 1 of figure 1.1 asserts that within the module named server.s, the expression 1.a (below) must be satisfied.

1.a. $(\text{in req\_q})(\text{Element}(m,c) \rightarrow \Diamond \text{Head}(m,c))$

Expression 1.a, in turn, asserts that within the component named req_q, the implication $(\text{Element}(m,c) \rightarrow \Diamond \text{Head}(m,c)$ must be true. We let $\text{Element}(m,c)$ indicate the presence of message m from client c in the queue. And we let $\text{Head}(m,c)$ indicate that (m,c) is at the head of the queue. Thus, expression 1 of figure 1.1 states that the component req_q within the component server.s is a *live* queue: it will move every enqueued element to the head. In other words, server s contains a component called req_q which possess a property that is often proper for queues. The second assertion of figure1.1 suggests that if a request reaches the head of the request queue in server.s, then the server will inevitably reach a state where a response to the client is in its response queue. This is an assertion about the server: the queues alone cannot ensure this property. The first two assertions raise our hopes about the correctness of the system in question, but the last assertion dashes these hopes. The last assertion claims that if the server does have (m,c) in its request queue, there is no implication that the server will inevitably have a response in its output queue. Appearances to the contrary, there is no contradiction here. The first two assertions make it clear that the server algorithm will generate a response. But these assertions do not say anything about whether or not the server algorithm will be allowed to progress. Progress is a

system property, not a property that the server can assure. The multi-level quality of these assertions is beyond the expressive range of the temporal logics.

Chapter 5 applies the m.p.r. arithmetic to the problem of systems with real-time constraints. We develop some simple techniques for measuring time in the transducer model, and show that we can define m.p.r. analogs of the modal modifiers of real-time temporal logic [31]. We then look at two examples, a fragment of the Futurebus+ arbiter, and a real-time priority queue. The Futurebus+ example illustrates methods for dealing with a real-time distributed algorithm. The priority queue section provides a detailed example of specification refinement: the process of developing a more detailed specification from an abstract specification, and then proving that the new specification *implements* the original.

Chapter 6 is concerned with the example of a fault tolerant broadcast protocol. Fault tolerance is one of the most important and hard to verify properties at the system-level. We formalize a quite sophisticated algorithm for fault tolerant message transfer over an Ethernet style broadcast network. The algorithm is taken from the network literature [9], rather than being synthesized for purposes of illustration.

The final chapter contains a conclusion and describes possible future research, including prospects for automation of proofs.

The remainder of this chapter situates the m.p.r. arithmetic within the formal methods literature.

1.5 Related literature

M.p.r. arithmetic is an example of a computational formalism, but derives a great deal from mathematical sources which have been ignored in the formal methods community. These sources include the primitive recursive arithmetic of Goodstein [20], and the algebraic automata products described by Gecseg [17]. Rather than attempt an exhaustive comparison between m.p.r. arithmetic and other proposed formal methods, this section will briefly touch on some related work in mathematics and computer science, and then will examine the relationship between m.p.r. arithmetic and temporal logic.

1.5.1 Sources and related formal methods

The basic technique for describing past state in the m.p.r arithmetic involves a precedence comparison between the i^{th} most recent instance of one type of transition, and the j^{th} most recent instance of a second type. For example, we say that an item be-

longs to a queue if the most recent enqueue transition is preceded by the most recent dequeue transition. This "counting backwards" technique is seemingly obvious, but is absent in the formal methods literature. Precedence comparison is, however, closely related to Rabin's definite regular languages [51], and threshold counting [4].

The m.p.r. techniques for specifying automata via modal functions are original, but are related to methods of Buchi and others in formal language theory [49, 34] and complexity theory (see [4] for a survey). Kripke [32] first showed how graph structures can be investigated with modal terms and deductions. Clarke and his colleagues [10, 54], Vardi [60], and Ostroff [46], among others, have used explicitly defined automata as semantic models for modal logics. That is, given an automaton M and an assertion Q of some modal logic, these researchers have considered methods of *checking* the truth of Q against M. We seek instead to replace explicit definitions of automata by symbolic definitions, and to reason about the symbolic definitions algebraically.

The primitive recursive arithmetic was developed by Goodstein [21, 20], and relies on the initial work of Peter [48] and Skolem [56]. Although the p.r. functions are well known to computer science, they have been mostly ignored by those working in computational logic and formal methods. We are not aware of previous modal formalisms based on p.r. functions, although Smorynski [57] describes an unrelated modal primitive recursive logic of provability.

While algebraic automata theory enjoyed a brief vogue in the late 1960's [18, 25], the major focus of interest was decomposition of state machines into feedback-free products. In the literature of state machine based computational formalisms [46, 24, 38, 22], the cartesian product of automata is about as sophisticated as it gets. Gouda [22] connects automata via unbounded message buffers, thus introducing a constricted and unrealistic view of how modules interact (unbounded buffers being rather few and far between in digital engineering). Lynch [38] defines Input/Output-Automata which are composed in such a way that the alphabet of the combined automaton is the union of the alphabets of the components, and state transition a causes all factors which include a in their alphabets to change state in parallel. Similar composition techniques are also used in CSP [27] and CCS [39]. This alphabet directed composition is very poorly suited to systems which connect components by means other than synchronous message transfer. And all three of these methods require elaborate conventions about fair scheduling. Harel's Statechart method [24] constitutes a rather more sophisticated attack on specification via composed automata. Statecharts provide a graphical notation in which related states and states

of concurrent systems can be grouped together to form meta-states. Unfortunately, Statcharts are based on a broadcast model of communication that is not appropriate for many of our target systems.

1.5.2 Comparison with temporal logic

M.p.r. arithmetic is to a great degree inspired by the temporal logics [50, 33]. The basic idea of temporal logic is to add modifiers to a propositional or predicate logic so that the dimension of time can be given formal expression. A proposition $\Box Q$ is true if and only if Q must be true "henceforth". A proposition $\Diamond Q$ is true if and only if Q must become true "sometime" in the future. The simple *and intuitive* nature of these modifiers has made temporal logic seem like a good basis for reasoning about state change. For example, $\Box(\text{Requesting(id)} \rightarrow \Diamond\text{Serving(id)})$ captures a desirable property of a resource allocator — that every request will eventually be served. Properties of liveness (something good will happen), safety (nothing bad will happen) and fairness can be given succinct and natural definitions in temporal logics [47, 52].

The precise meaning of temporal assertions is usually given in terms of computation paths or trees [50]. A computation path is a sequence of states where state σ_{i+1} is considered to be the state reached one time unit (or one program statement) after state σ_i. Computation trees allow for choice, σ may have several possible successor states. States are generally *snapshots* of the contents of store or, more formally, assignment functions mapping propositions and variables into appropriate domains. For example, if σ is a state, then $\sigma(x)$ is the value of variable x in σ, and $\sigma(Q) = \text{True}$ iff Q is a true proposition in σ. The branching temporal logic [5, 16] is used to reason about (possibly infinite) trees of states. Suppose that σ is the current state. A proposition $\bigcirc Q$ is true iff $\sigma'(Q) = \text{True}$ for each child σ' of σ — if Q must become true in the next state. A proposition $\Box Q$ is true iff $\sigma(Q) = \text{True}$ for every descendent of σ (including σ itself). A proposition $\Diamond Q$ is true iff there is an integer k so that every path of length k rooted at σ must contain at least one state σ' with $\sigma'(Q) = \text{True}$. There is also a binary temporal quantifier until so that Q until Q′ is true iff every path rooted at the current state must either keep Q true or must first make Q′ true.

One can sense that temporal logic is not a complete answer to the problem of formal verification from the large body of literature suggesting extensions to temporal logic. Ostroff has added clocks and history variables in order to describe real-time [46]; Kooymans has added real-time subscripts [30, ?]; Wolper has added regular expressions [61]; Lichtenstien [50] has added past tense quantifiers; Pnueli has added

history variables [50]; and Moszkowski has added *interval quantifiers* [42]. While there is something valuable in each of these extensions, and in others not listed here, each extension complicates the underlying semantics, and none seems sufficient in itself for the full range of problems encountered at the system-level.

Examination of efforts to apply temporal logic to complex systems reveal several basic weakness.

- "Next state" seems incompatible with composition. For example, Clarke *et al* [7] are forced to verify circuits at the gate level in order to let "next state" mean "after one gate delay." A hierarchical verification, in which sub-circuits consisting of several gates could be treated as units, is not possible in the "next state" framework because the "next state" of components might not match the "next state" of the composite system. Similar concerns prevent use of a more realistic model in which gate delays are not uniform. Some researchers have suggested dropping $\bigcirc$ from the temporal logic in order to avoid precisely this problem [35, 11]. But this approach requires us to abandon hope of verifying detailed timing constraints.

- Multi-level reasoning is not supported. Even without $\bigcirc$, composition is not satisfactory. Temporal logic provides a *flat* view of computation, and the type of multi-level assertion given in figure 1.1 above is not possible within the temporal framework.

- Algorithms must be defined outside the logic. The temporal logic provides an awkward notation for detailed definition of algorithms. Thus, temporal specifications often make use of programming language notation, and must resort to arguments about program statement labels (e.g. [10]). In contrast, the m.p.r. arithmetic uses the same functional language to express both very high level temporal properties, and very detailed specifications of algorithms.

- Causality is obscured by the semantics. Temporal logic requires extensive axiomatization of the dynamic behavior of functions and variables because it does not capture the causality of discrete systems. The values of state variables in actual computational systems change only in response to some event or the passage of time. In m.p.r. arithmetic we define state sensitive functions so as to emphasize this causal sensitivty. If f depends on events a and b, then we know that c will not alter the value of f. This causality is not captured by temporal logic semantic structures, and consequently, the dynamic behavior of each

state variable requires extensive axiomatization or the introduction of history variables. But history variables defeat the purpose of the logic by making the context visible.

- Proofs tend to be very tedious. Boute [6] has argued that formal computational logics are inherently awkward and opaque. He suggests that because formal logics are outside of the framework of standard mathematics, they provide a stilted and cryptic basis for deduction. This certainly seems to be the case for temporal logics. In order to reason about numbers within the temporal logic one must use a many sorted predicate temporal logic, and import *domain axioms* for the variables of integer sort. We are not persuaded that reasoning about finite state systems requires these methods.

In sum, we find temporal logic to provide an appealing, but flawed, basis for reasoning about systems-level computation. The m.p.r. arithmetic provides an alternative, constructive semantics for the temporal operators, plus a finer grain of resolution, multi-level semantics, and a more causal perspective.

1.6 Summary

The problem of formal analysis of system-level computation motivates this work. To address this problem we introduce a formal method based on finite state machine theory, modal techniques, and primitive recursive arithmetic. In this chapter we argue that the rich semantics and appealing mathematical properties of algebraic automata products provide an expressive and intutive semantic basis. We sketch the outlines of a language of functions which permits abstract specification of automata and automata products, and we indicate that there are strong methods available for verifying properties of these specifications.

CHAPTER 2

SYNTAX AND SEMANTICS

The first section of this chapter provides an intuitive introduction to the m.p.r. arithmetic. The second section defines transducers, transducer products and relative precedence within traces. The third section reviews the class PR of primitive recursive functions. The final section introduces the class MPR of m.p.r. functions and a functional $\rho : \mathsf{MPR} \to \mathsf{PR}$ which defines the value of m.p.r. functions. Readers not interested in the mathematical underpinnings of the arithmetic are invited to skip all but the first section.

Sequence notation. This chapter involves a great deal of sequence manipulation. Unfortunately, there are no standard names for sequence functions. Thus, we must explicitly define the notation that will be used here. Let $\langle\rangle$ denote the empty sequence, and let $\langle a_1, ..., a_n\rangle$ denote the named sequence of length n. The concatenation of sequences is denoted by juxtaposition or the symbol '$\cdot$', whichever seems clearer in the context:

$$\begin{aligned}\langle a_1, ..., a_n\rangle\langle b_1, ..., b_k\rangle &= \langle a_1, ..., a_n\rangle \cdot \langle b_1, ..., b_k\rangle \\ &= \langle a_1, ..., a_n, b_1, ..., b_k\rangle.\end{aligned}$$

We let $(\langle a_1, ..., a_n\rangle)_i = a_i$ if $0 < i \leqslant n$ and 0 otherwise. Let au abbreviate $\langle a\rangle u$ and let ua abbreviate $u\langle a\rangle$ when it is clear that "a" represents a single element and "u" represents a sequence. Finally, the language of sequences A^* consists of $\langle\rangle$ and ua for all $u \in A^*$ and $a \in A$. These, and other useful string functions are listed, for convenience, in figure 2.1. More formally, strings within the primitive recursive arithmetic are *encoded* in natural numbers. In practice the only difference this makes is that we do not require a type theory: all variables range over natural numbers.

Concatenation

$\langle a_1, ..., a_n\rangle\langle b_1, ..., b_k\rangle = \langle a_1, ..., a_n, b_1, ..., b_k\rangle$

$\langle a_1, ..., a_n\rangle \cdot \langle b_1, ..., b_k\rangle = \langle a_1, ..., a_n, b_1, ..., b_k\rangle$

Left Appending

$au \stackrel{def}{=} \langle a\rangle u$

Right Appending

$ua \stackrel{def}{=} u\langle a\rangle$

Length

$\mathrm{length}(\langle\rangle) \stackrel{def}{=} 0, \mathrm{length}(wa) \stackrel{def}{=} 1 + \mathrm{length}(w)$

Last element of a sequence

$\mathrm{tail}(\langle\rangle) \stackrel{def}{=} 0, \mathrm{tail}(aw) \stackrel{def}{=} a$

First element of a sequence

$\mathrm{head}(\langle\rangle) \stackrel{def}{=} 0, \mathrm{head}(wa) \stackrel{def}{=} a$

Left truncation

$\mathrm{ltrunc}(\langle\rangle) \stackrel{def}{=} \langle\rangle, \mathrm{ltrunc}(wa) \stackrel{def}{=} w$

Right truncation

$\mathrm{rtrunc}(\langle\rangle) \stackrel{def}{=} \langle\rangle, \mathrm{rtrunc}(aw) \stackrel{def}{=} w$

Indexing[a]

$(\langle\rangle)_i \stackrel{def}{=} 0, (u)_0 \stackrel{def}{=} 0, (aw)_1 \stackrel{def}{=} a, (aw)_{i+2} \stackrel{def}{=} (w)_{i+1}$

Pumping

$(u)^0 \stackrel{def}{=} \langle\rangle, (u)^{i+1} \stackrel{def}{=} u \cdot (u)^i$

Prefix

$\mathrm{prefix}(\langle\rangle, i) \stackrel{def}{=} \langle\rangle, \mathrm{prefix}(aw, 0) = \langle\rangle, \mathrm{prefix}(aw, i+1) \stackrel{def}{=} \langle a\rangle \mathrm{prefix}(w, i)$

Prefix membership

$v \prec u \stackrel{def}{=} (\exists i \leqslant \mathrm{length}(u))\mathrm{prefix}(u, i) = v$

Figure: 2.1 Sequence functions

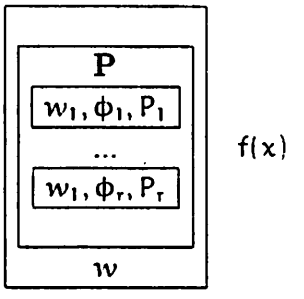
[a]In defiance of Dijskstra, we index sequences starting at 1 instead of 0.

2.1 The modal arithmetic

The m.p.r. functions are obtained by adding seven new *initial* functions and two new function composition rules to the class of primitive recursive functions. The initial functions fall into two groups. The first group contains constant functions that describe system constants such as the alphabet. The second group consists of state dependent initial functions which depend on whether or not a transition is

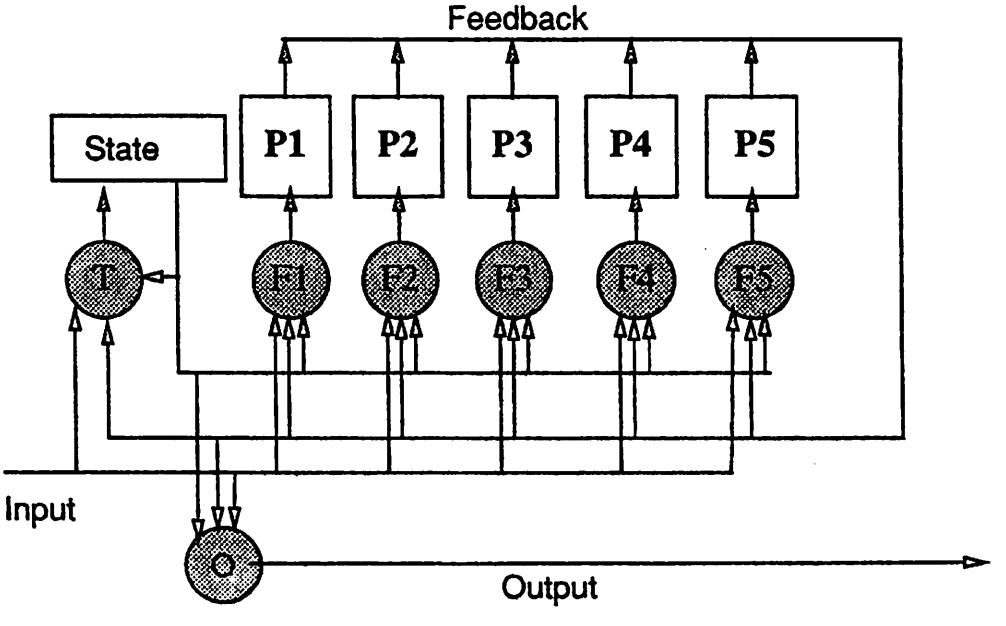
enabled in the current state, and on the ways in which components of a composite system are interconnected. More sophisticated functions are built from these using the composition rules of the primitive recursive functions, and also using the new rules which allow for definition of functions that will be evaluated in future contexts, and in the contexts of factors.

In general the process of evaluating an m.p.r. function f(x) in a context (P, w) can be pictured as follows.



The large box to the left of f(x) depicts the context, containing the product form transducer P, and its trace w. The smaller boxes inside P depict the factor transducers $P_1, ..., P_r$, their associated feedback functions $\phi_1, ..., \phi_r$ and traces $w_1, ..., w_r$. A transducer is called *flat* if it contains no factors.

Figure 2.2 provides a more detailed picture of a product form transducer.



P1–P5 are factors, T is a transition function, and F1–F5 are feedback functions.

Figure: 2.2 A product form transducer.

The state of a product form transducer is composed from the "top-level" state and the states of the factors, which may also be in product form. The "top-level" state represents that part of the state which is new to the product transducer — not derived directly from the states of the factors. For example, the top-level state of a transducer representing a circuit may track the propagation delays of signals traveling between the devices represented by the factors. The *product state set* of a flat transducer is the same as its top-level state set. But the product state set of a transducer with factors is the cartesian product of the top-level state, and the product state sets of the factors. Thus, a product state of a product transducer is a tuple $(s, s_1, ..., s_r)$. In order to model encapsulation, the product form transducer is not permitted to use the raw state of the factors when it generates output, changes state, or generates input for the factors. Instead, the product transducer is restricted to using the outputs of the factors and the top-level state. We call a tuple $(s, o_1, ..., o_r)$ a *configuration* when s is a top-level state and each o_i is an output symbol from P_i. Every product state $(s, s_1, ..., s_r)$ corresponds to exactly one configuration $(s, o_1, ..., o_r)$, where each o_i is the output generated by factor P_i in state s_i. Thus, there are at most as many possible configurations as there are product states. In general, there will be fewer configurations than product states.

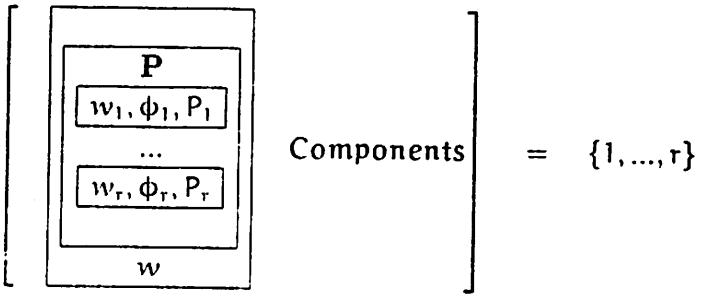
The values of m.p.r. functions in a context (P, w) will depend on the structure of P, on the relative order of transitions within w, on the configuration reached by following w from the initial state, and on the states of the factors. We let $\Delta(P, w)$ denote the configuration to which w drives P; we let $\Delta(P, w)$ be undefined if either: w drives causes an undefined state transition of the top level transition function, or the factor traces induced by w cause one of the factors to enter an undefined state. We let $\mathcal{L}(P)$ be the language of enabled traces w — the traces w so that $\Delta(P, w)$ is defined. Thus, $\mathcal{L}(P)$ is the set of possible behaviors for P.

2.1.1 The initial functions

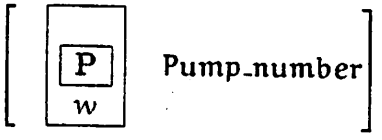
The initial constant functions There are three m.p.r. functions which give us the basic constants of the context. The function Alphabet returns a set containing all the transition symbols of P. By convention, we will generally omit the empty parenthesis () for functions with no arguments — writing Alphabet in place of Alphabet().

$$\left[\boxed{\begin{matrix}\mathbf{P} \\ w\end{matrix}} \quad \text{Alphabet} \right] \quad = \quad \text{the alphabet of } \mathbf{P}$$

The function Components returns a set containing names for each factor of P. Thus, evaluation of Components requires us to "open up" P and look at its structure.



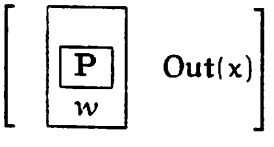
The value of the function Pump_number = the size (cardinality) of the product state set.



In practice, the actual value of Pump_number is not of interest, but the function Pump_number is useful as a symbolic representation of the value. For example, an output function Out(x) can have a range of at most Pump_number values, because the value of Out(x) depends only on the configuration.

The initial state functions and trace languages. There are four basic m.p.r. functions which can test properties of the current state: Out, enable, f_effect, and precedes.

- Each transducer will be associated with an output function which generates an output, depending solely on the current configuration. Generally the output will be a tuple of values. For example, the output of a bus interface will be a tuple consisting of outputs to each bus wire. Thus, Out(x) is the x^{th} element of the current output tuple.



- The function f_effect(a, c) defines the path which will be induced for factor c if an a transition is traversed from the current configuration. The value of f_effect reflects the value of the feedback functions. When f_effect(a, c) = u, the feedback function ϕ_c will generate the sequence u for factor c if the product transducer accepts an a. For example, in a circuit which monitors k

serial lines and has k factors representing flip flops, a transition $sense.\langle x_0, \ldots x_{k-1}\rangle$ might induce each flip flop FF_i to latch x_i — $f\_effect(sense.\langle x_0, \ldots, x_{k-1}\rangle, FF_i) = \langle latch.x_i\rangle$. Note that $f\_effect(a, c)$ is a sequence over the alphabet of the factor transducer named c, thus the alphabets of the factors can be completely distinct from the alphabet of the product transducer. When $f\_effect(a, c) = \langle\rangle$, factor c will not change state due to a.

$$\left[\begin{array}{c} \boxed{\begin{array}{c}\mathbf{P}\\ \boxed{\phi_1, P_1}\\ \ldots\\ \boxed{\phi_r, P_r}\\ \\ w\end{array}} \end{array}\right] \quad f\_effect(a, c) \quad = \quad \text{the feedback induced in the current configuration}$$

- The boolean function $enable(a) = 1$ iff an a transition is *enabled* in the current configuration — iff a can be appended to w to obtain a new trace which does not drive P into an undefined configuration. Note that if $f\_effect(a, c)$ drives factor c into an undefined state, then a cannot be enabled.

$$\left[\begin{array}{c} \boxed{\begin{array}{c}\boxed{\mathbf{P}}\\ w\end{array}} \end{array}\right] \quad enable(a) \quad = \begin{cases} 1 & \textbf{if } wa \in \mathcal{L}(P); \\ 0 & \textbf{otherwise.} \end{cases}$$

- Given a trace $w = \langle a_1, \ldots, a_n\rangle$ we say that a_i represents an event that preceded a_j iff $i < j$. The "most recent" event represented in w is given by a_n, the "second most recent" is given by a_{n-1}, and the " i^{th} most recent event" is given by $a_{n-(i-1)}$. It is convenient to have an imaginary event a_0 to represent events which have not yet occurred, i.e., the k^{th} most recent event for $k > n$. We can also consider the "i^{th} most recent b event" for some b in the alphabet to be that a_i so that $a_i = b$ and there are exactly i events $a_k = b$ with $k \geqslant j$. Again, we will find it convenient to let a_0 stand for the k^{th} most recent b if there are less than k b's in the sequence. The boolean function $precedes(a, i, b, j)$, is true iff the i^{th} most recent a transition recorded in the current trace *preceded* the j^{th} most recent b transition recorded in the current trace. For example, in a site on a computer network, we might test to see if a message has been sent more recently than its acknowledgment has been received, by evaluating $precedes(receive.ack_m, 1, send.m, 1)$. In a real-time circuit, we might test to see if at least k time units have passed since the logic level was most recently raised by evaluating $precedes(raise, 1, tick, k)$. In order to evaluate precedes

on a trace, we can define $Place(w, a, i)$ to be 0 if there are fewer than i a's in w, or if $a = 0$, and let $Place(a, i) = j$ such that $(w)_j = a$ and $\sum_{k=1}^{n}[(w)_k = a] = i$, otherwise. By convention 0 denotes the null transition which leaves state unchanged, and we find it convenient to just ignore the 0's in the trace. Thus, $precedes(a, i, b, j)$ will be true of w iff $Place(w, a, i) < Place(w, b, j)$.

$$\left[\boxed{\begin{array}{c}\mathbf{P}\\ w\end{array}}\ precedes(a, i, b, j)\right] = \begin{cases} 1 & \textbf{if } Place(w, a, i) < Place(w, b, j); \\ 0 & \textbf{otherwise.}\end{cases}$$

We can easily build quite sophisticated state predicates from precedence comparisons, but the comparisons have a particularly simple algebraic interpretation. The integers i and j in a comparison between the i^{th} most recent a and the j^{th} most recent b are called the precedence indexes of the comparison. Whenever the precedence indexes of a state expression are bounded by k, only the k most recent transitions of each type can effect state. Thus, the language of sequences which satisfy a particular precedence constraint must be regular.

We now define two important functions from *precedes*. The boolean function $Past(a, i) = 1$ iff there are at least i a's in the current trace. The boolean function $Initial = 1$ iff the current trace is $\langle\rangle$.

Definition 2.1: $Past(x, y)$

$$Past(x, y) \stackrel{\text{def}}{=} precedes(0, 1, x, y)$$

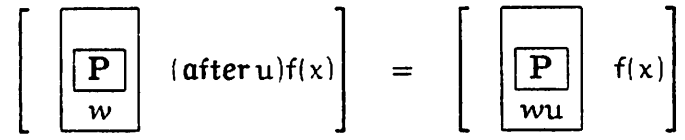
Definition 2.2: $Initial$

$$Initial \stackrel{\text{def}}{=} \neg(\exists a \in Alphabet)Past(a, 1)$$

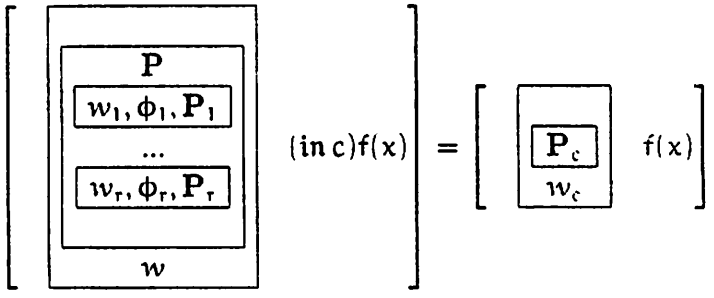
2.1.2 The composed functions

We can build quite sophisticated state functions from the seven m.p.r. initial functions, the p.r. functions, substitution ($f(x) = h(g(x))$), and primitive recursion. We also have two *modal* function modifiers (functionals) which allow us to define functions that are to be evaluated either in a future state, or in the context of a sub-system (factor). For every m.p.r. function $f(x)$ we have a function $f'(u, x) \stackrel{\text{def}}{=} (\text{after } u)f(x)$, and a function $f''(c, x) \stackrel{\text{def}}{=} (\text{in } c)f(x)$. For any sequence of state transitions u, the value of $(\text{after } u)f(x)$ is the value of $f(x)$ in the *future* state reached by following u from the

current state. For any component name c, the value of (in c)f(x) is the value of f(x) in the context of the factor named c.



Informal definition of after.



Informal definition of in.

Figure: 2.3 Informal definition of the modal functionals.

2.1.3 Applications of after

Although after may appear to be a very primitive operation, it can be the basis of quite sophisticated concepts of liveness, timeliness, eventuality, and safety. We can first use after to extend enable and f_effect to sequences, so that we can find out if a sequence u is enabled, and the effect of a sequence u on a component.

Definition 2.3: Extension of enable and f_effect .

$$\begin{aligned}
&\mathsf{enable}^*(\langle\rangle) \stackrel{\mathsf{def}}{=} 1\\
&\mathsf{enable}^*(u a) \stackrel{\mathsf{def}}{=} \mathsf{enable}^*(u) * (\mathsf{after}\, u)\, \mathsf{enable}(a)\\
&\mathsf{f_effect}^*(\langle\rangle, c) \stackrel{\mathsf{def}}{=} \langle\rangle\\
&\mathsf{f_effect}^*(a u, c) \stackrel{\mathsf{def}}{=} \mathsf{f_effect}(a, c) \cdot (\mathsf{after}\, u)\, \mathsf{f_effect}^*(u, c)
\end{aligned}$$

In the sequel we will often abuse notation, writing enable(u) for enable*(u), and relying on the type of the argument to disambiguate. We will use a similar convention for f_effect and f_effect*. It will also be convenient to write (after a)f in place of (after⟨a⟩)f when this can be done without confusion.

We can now, informally, test to see if a function is *henceforth* true.

Definition 2.4: Informal version of $\Box$

$$\Box f(\vec{x}) > 0 \Leftrightarrow (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u) f(\vec{x}) > 0).$$

Thus, $\Box f(x)$ is true, iff $f(x)$ is true in all reachable configurations (including the current configuration).

We can also say that sometime in the future $f(x)$ can take on a value k, and then remain equal to k indefinitely as follows.

$$(\exists v, u, z)(\forall n)\left(\begin{array}{l} \mathbf{enable}(v \cdot \underbrace{u \cdot \ldots u}_{n \text{ times}} \cdot z) \\ \bigwedge(\mathbf{after}\, v \cdot \underbrace{u \cdot \ldots u}_{n \text{ times}} \cdot z) f(x) = k) \end{array}\right)$$

In chapter 4 we show how to define these and other interesting path properties, using the pumping properties of regular languages and Pump_number to get rid of the unbounded quantification.

2.2 Transducers and traces

Product form transducers are classical transducers presented as a product of transducers. Each of the factors represent a component sub-system and there is additional structure representing the coordination of these sub-systems. Since the factors are also product form transducers, we can consider product form transducers as trees with leaves being transducers representing systems that are atomic (systems with no components).

We define product form transducers inductively. First we define the class $\mathbf{P}_0$ of flat product form transducers — those with no factors. The elements of this class are essentially standard Moore machines [40, 29]. The class $\mathbf{P}_{i+1}$ of transducers consists of those transducers containing at least one factor belonging to $\mathbf{P}_i$, and no factors of belonging to $\mathbf{P}_j$ for $j > i$. The class $\mathbf{P}$ is the union over all $\mathbf{P}_i$.

2.2.1 Flat transducers

It is convenient to consider all alphabets to consist of the integers $\{1, \ldots, k\}$ for some k, and to consistently reserve 0 for the null transition that leaves state unchanged.

Definition 2.5: The flat product form transducers
The class $\mathbf{P_0}$ of flat product form transducers is the smallest set containing all tuples of the form:

$$P = (A, S, start, O, \lambda, \delta)$$

$A = \{1, ...n\}$ is a finite transition alphabet,

$O = \{1, ...k\}$ is a finite output alphabet,

$S = \{1, ...h\}$ is a finite state set,

$start \in S$ is a distinguished start state,

$\lambda : S \rightarrow O$ is an output function,

$\delta : S \times A \rightarrow S$ is a transition function, with $\delta(s, 0) = s$.

We can define a function $\Delta : \mathbf{P_0} \times A^* \rightarrow S$ so that $\Delta(P, w)$ is the state to which w drives P from the initial state. Thus, given a transducer P, and a trace w, we can derive the current state of P — $\Delta(P, w)$.

Definition 2.6: Δ for flat transducers

$$\Delta(P, \langle\rangle) \stackrel{def}{=} initial$$
$$\Delta(P, wa) \stackrel{def}{=} \delta(\Delta(P, w), a)$$

2.2.2 Product form transducers

A general product form automaton P is also defined as a tuple including a state set, input and output alphabets, and transition and output functions. But, each of these product form transducers also contains a tuple of *factor* product form transducers $F = (P_1, ..., P_r)$, and a tuple of *feedback functions*. Factors are connected by making the input to each factor depend on the input to the product form automaton, the state of the product form automaton, and the outputs of all of the factors. When a product form automaton accepts a single input symbol, each factor is provided with a sequence of 0 or more input symbols, representing the parallel activity of all the components. For example, suppose we have a product form automaton representing a network of computers, and containing factor transducers representing each individual site. A single state transition representing a message transfer might induce receive state transitions for some of the factors, a send transition for one factor, and no transitions for factors representing sites not involved in the transfer.

Notation. When we define a product form automaton, we need a convenient notation to distinguish elements of the product, such as alphabet and state set, from the elements of its factors. Thus, we write δ to refer to the transition function of P,

and write $\delta.i$ to refer to the transition function of the i^{th} factor of P. Similarly, if O is the output alphabet of a transducer, then O.i is the output alphabet of factor P_i.

Definition 2.7: The class of product form transducers
The class **P** of *product form transducers* is the infinite union of the classes $\mathbf{P}_i$ for $i \geq 0$. Each class $\mathbf{P}_{l-1}$ of product form transducers is the smallest set containing all tuples of the form:

$$P = (A, O, S, start, \lambda, \delta, F, \Phi)$$

$A = \{1, ...n\}$ is a finite transition alphabet,

$O = \{1, ...k\}$ is a finite output alphabet,

$S = \{1, ...h\}$ is a finite state set,

$start \in P.S$ is a distinguished start state,

$F = (P_1, ...P_r)$ is a tuple of product form transducers, so that $l = max\{l' : (\exists i)P_i \in \mathbf{P}_{l'}\}$.

$\lambda : S \times O.1... \times O.r \rightarrow O$,

$\delta : S \times O.1... \times O.r \times A \rightarrow S$ is a transition function, with $\delta(s, o_1, ..., o_r, 0) = s$.

$\Phi = (\phi_1, ...\phi_r)$ is a tuple of feedback functions: $\phi_i : S \times O.1... \times O.r \times A \rightarrow (A.i)^*$, with $\phi(s, o_1, ..., o_r, 0) = \langle\rangle$.

Note that the transition and output functions can only use the output of the factors, and do not see the internal state of factors.

The product state set of a flat transducer P, is the state set of P. If P is not flat, then the product state set of P, denoted PS(P) is given as follows:

Definition 2.8: The product state set PS(P)

$$PS(P) \stackrel{def}{=} S \times \prod_{i=1}^{r} PS(P_i).$$

A *configuration* is a tuple $\mathbf{c} = (s, o_1, ..., o_r) \in S \times O.1... \times O.r$. The start configuration is the tuple consisting of start and the the outputs of each factor in its own start configuration. Since the start configuration of a flat transducer is start, the recursion must terminate. Let $st = (start, \lambda.1(st.1), ..., \lambda.r(st.r))$ denote the start configuration.

After an a transition, we will get a new configuration $\mathbf{c}' = (\delta(\mathbf{c}, a), o'.1, ...o'.r)$ where each $o'.i$ is of the configuration which P_i reaches by following $\phi_i(c, a)$ from its own current configuration.

Intuitively, we intend that when a product form automaton accepts a transition a, each factor will traverse transition sequence $\phi_i(c, a)$. We formalize the concept of current configuration by defining two functions. The first is a map Δ so that $\Delta(P, w)$ is the configuration of P after it has accepted the path w as input. The second is ϕ_i^*, the reflexive transitive closure of ϕ_i. We define Δ inductively on the classes comprising

P. For P in $\mathbf{P}_0$, we have already defined $\Delta(P, w)$. For P in $\mathbf{P}_{1..}$ we need to also use the configurations of the factors. We assume that $\Delta.i$ is defined for each of the factors. Let **c** be a configuration. Then we can define ϕ_i^* as follows.

Definition 2.9: Reflexive, transitive closure of ϕ

$$\phi_i^*(\mathbf{c}, \langle\rangle) \stackrel{\text{def}}{=} \langle\rangle$$
$$\phi_i^*(\mathbf{c}, aw) \stackrel{\text{def}}{=} \phi_i(\mathbf{c}, a)\phi_i^*(\delta(\mathbf{c}, a), \lambda.1(\Delta.1(a)), ...\lambda.r(\Delta.r(a)), w))$$

The empty sequence $\langle\rangle$ induces an empty sequence for the factors. Each a transition in configuration **c** will take the transducer to a new configuration $(\delta(\mathbf{c}, a), \lambda.1(\Delta.1(a)), ...\lambda.r(\Delta.r(a)), w))$. Thus, the sequence induced by aw in configuration **c** is obtained by concatenating the sequence induced by a to the sequence induced by w in the configuration resulting from a. The function $pfactor(w, i)$ gives the sequence induced for i by w from the start configuration.

Definition 2.10: pfactor and Δ for product form transducers

$$pfactor(w, i) \stackrel{\text{def}}{=} \phi_i^*(\mathbf{st}, w)$$

$$\Delta(P, \langle\rangle) \stackrel{\text{def}}{=} (start, start.1, ..., start.r)$$
$$\Delta\ (P, wa) \stackrel{\text{def}}{=}$$
$$(\delta(\Delta(P, w), a), \lambda.1(\Delta.1(P_1, pfactor(wa, 1))), ..., \lambda.r(\Delta.r(P_r, pfactor(wa, r)))$$

Note that $\Delta(P, w)$ is defined only if $\Delta.i(P_i, \phi_i^*(\mathbf{st}, w))$ is also defined. This means that any trace of a product form transducer will not drive any of the factors into undefined states. We can now, formally, define the language of traces $\mathcal{L}(P)$

Definition 2.11: The trace language $\mathcal{L}(P)$

$$\mathcal{L}(P) \stackrel{\text{def}}{=} \{w \in A^* : \Delta(P, w) \text{ is defined } \}$$

2.2.3 Transition precedence.

In order to evaluate precedes we need to be able to translate an expression $precedes(a, i, b, j)$ into an assertion about the current trace. First we define a function $place(w, a, i)$ which describes the position of (a, i) in w. Intuitively, $place(w, a, i)$ ranks the relative order of the i^{th} most recent (rightmost) a transition in w, with higher values indicating "more recent". We find the following conventions useful:

- Recall that the alphabets of state machines are sets $\{1, ...k\}$, and 0 is reserved for the null transition which leaves state unchanged. It is convenient, to con-

sider all null transitions as less recent than any non-null transitions, so we let $\mathrm{place}(w, 0, i) = 0$.

- The rightmost a transition in w is considered to be the 1^{st} most recent a transition, so $\mathrm{place}(w, a, 1)$ is the greatest j such that $(w)_j = a$.
- If there are fewer than i a transitions in w, then $\mathrm{place}(w, a, i) = 0$, the null transition.

Definition 2.12: Place

The "place" of a pair (a, i) in a trace.

$$\mathrm{place}(\langle\rangle, a, i) \stackrel{\mathrm{def}}{=} 0$$

$$\mathrm{place}(ub, a, i) = \begin{cases} 0 & \textbf{if } a = 0 \text{ or } i = 0 \\ \mathrm{length}(ub) & \textbf{else if } a = b \bigwedge i = 1 \\ \mathrm{place}(u, i-1) & \textbf{else if } a = b \\ \mathrm{place}(u, a, i) & \textbf{otherwise.} \end{cases}$$

We say that (a, i) precedes (b, j) in w iff $\mathrm{place}(w, a, i) < \mathrm{place}(w, b, j)$.

Definition 2.13: RelOrder

$$\mathrm{RelOrder}(w, a, i, b, j) \stackrel{\mathrm{def}}{=} \begin{cases} 1 & \textbf{iff } \mathrm{place}(w, a, i) < \mathrm{place}(w, b, j); \\ 0 & \textbf{otherwise.} \end{cases}$$

It is important to realize that each expression $\mathrm{RelOrder}(w, a, i, b, j)$ defines a finite indexed congruence on A^*. Define the congruence as follows.

Definition 2.14: A congruence on RelOrder

$$u \equiv v \bmod (a, i, b, j) \Leftrightarrow (\forall x, y)\mathrm{RelOrder}(xuz, a, i, b, j) = \mathrm{RelOrder}(xvz, a, i, b, j)$$

Definition 2.15: The congruence classes induced by RelOrder

$$[w]_{\cdot a,i,b,j)} = \{u : u \equiv w \bmod (a, i, b, j)\}$$

Theorem 1: For any fixed (a, i, b, j) the set $\{[w]_{\cdot a,i,b,j)}\}$ contains less than $\sum_{k=0}^{i+j} n^k$ distinct elements, where n is the cardinality of A.

Proof. Let $\mathrm{length}(w) > i+j$. If w contains any elements that are equal to neither a or b, those elements can be discarded to obtain a shorter, congruent sequence. If w contains only a's and b's, then there are either more than i a's or more than j b's. If there are more than i a's we can discard the leftmost a to obtain a shorter, congruent sequence, and similarly for the b's. Thus any sequence of length greater than $i + j$ must be congruent to a shorter sequence.

The Myhill-Nerode theorem [43, 44, 29] states that any language which is defined as the union of finite indexed congruence relations is regular. Thus theorem 1 implies that the language $L \subset A^*$ where $L = \{w : RelOrder(w, a, i, b, j) = 1\}$ is a regular language. In fact, L is a particularly simple type of regular language, an aperiodic language. A language is aperiodic iff there is some t so that $wa^{t+i}v$ is in the language iff wa^tv is in the language [49]. This condition, obviously, holds in L.

2.3 Review of the Primitive Recursive Functions

The class of primitive recursive functions were first described by Skolem [56] and Godel [19]. Peter [48] has shown that a great deal of algebra and number theory is primitive recursive. Addition, multiplication, iterated summation and product, finite sets and sequences, factorization, bounded quantification, and the logical connectives $(\wedge, \vee, \neg \rightarrow)$ are all p.r. This may be surprising because the p.r. functions can be defined very simply.

Definition 2.16: The primitive recursive functions
The class PR of the primitive recursive functions consists of all the functions which can be generated after a finite number of steps using the following rules[a].

1. **Zero:** $f(\vec{x}) \stackrel{def}{=} 0$,
2. **Successor:** $f(x) \stackrel{def}{=} x + 1$,
3. **Projection:** $f(y, \vec{x}) \stackrel{def}{=} \vec{x}_y$
4. **Substitution:** $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}), ...g_n(\vec{x}))$,
5. **Primitive Recursion:** $f(0, \vec{x}) \stackrel{def}{=} g(\vec{x}), f(r+1, \vec{x}) \stackrel{def}{=} h(r, \vec{x}, f(r, \vec{x}))$,

[a]The rules given here, and the form of the presentation are from [57].

Each equation of the form $f(\vec{x}) \stackrel{def}{=} E(\vec{x})$ is called a *defining equation.* Any function f can be evaluated on an argument m by simply rewriting the expression repeatedly according to its defining equations. We will assume in this work that the name of a function is somehow associated with its defining equations. That is, if we are given f, we will be able to determine the defining equations of f. If we wanted to be formal about it, we could let the sequence of defining equations *be* the name of the function (c.f. [14]). However, such formality would be excessive here.

We can easily define the basic functions of arithmetic as p.r. functions. For example, we define addition, multiplication, and exponentiation using recursion:

$$0 + y \stackrel{\text{def}}{=} y, (x+1) + y \stackrel{\text{def}}{=} (x + (y+1))$$
$$0 * y \stackrel{\text{def}}{=} 0, (x+1) * y \stackrel{\text{def}}{=} (x * y) + y$$
$$x^0 \stackrel{\text{def}}{=} 1, x^{(y+1)} \stackrel{\text{def}}{=} x * (x^y)$$

Since we are concerned with functions over the non-negative integers, we cannot use standard subtraction. Instead, we define *cutoff* subtraction, so that $x \dot{-} y = 0$ if $x < y$.

$$\mathsf{Pred}(0) \stackrel{\text{def}}{=} 0, \mathsf{Pred}(x+1) = x$$
$$x \dot{-} 0 \stackrel{\text{def}}{=} x, x \dot{-} y + 1 \stackrel{\text{def}}{=} \mathsf{Pred}(x) \dot{-} y$$

We will commonly use *p.r. functionals*, mappings of the form $\alpha : \mathsf{PR}^n \rightarrow \mathsf{PR}$ to abbreviate complex definitions, or to describe classes of similar functions. Summation and interated product are classical examples of p.r. functionals.

$$\sum_{i=0}^{0} f(i, \vec{x}) \stackrel{\text{def}}{=} 1,$$
$$\sum_{i=0}^{k+1} f(i, \vec{x}) \stackrel{\text{def}}{=} f(k+1, \vec{x}) + \sum_{i=0}^{k} f(i, \vec{x})$$

$$\sum_{i=i-1}^{k} \stackrel{\text{def}}{=} \sum_{i=0}^{k \dot{-} (i+1)} f(i+j, \vec{x})$$

$$\prod_{i=0}^{0} f(i, \vec{x}) \stackrel{\text{def}}{=} 0,$$
$$\prod_{i=0}^{k+1} f(i, \vec{x}) \stackrel{\text{def}}{=} f(k+1, \vec{x}) * \prod_{i=0}^{k} f(i, \vec{x})$$

$$\prod_{i=i+1}^{k} \stackrel{\text{def}}{=} \prod_{i=0}^{k \dot{-} (i+1)} f(i+j, \vec{x})$$

In these examples, $\sum$ and $\prod$ are functionals, and $(\sum f)$ and $(\prod f)$ are p.r. functions. Bounded quantification is also defined in terms of p.r. functionals. In the definitions of the quantifier functionals, we make use of the function $\mathsf{sgn}(x)$ which reduces any value to a boolean value. We will generaly consider non-zero values to be *true*, and 0 to be *false*. The function sgn will be useful for converting arbitrary values to boolean values.

$$\mathsf{sgn}(0) \stackrel{\text{def}}{=} 0, \mathsf{sgn}(x+1) \stackrel{\text{def}}{=} 1$$
$$(\forall x < 0) f(x, \vec{z}) \stackrel{\text{def}}{=} 1$$
$$(\forall x < y+1) f(x, \vec{z}) \stackrel{\text{def}}{=} \mathsf{sgn}(f(y, \vec{x})) * (\forall x < y) f(x, \vec{z})$$
$$(\exists x < 0) f(x, \vec{z}) \stackrel{\text{def}}{=} 0$$
$$(\exists x < y+1) f(x, \vec{z}) \stackrel{\text{def}}{=} \mathsf{sgn}(f(y, \vec{x}) + \mathsf{sgn}((\exists x < y) f(x, \vec{z}))$$

The only glaring omission from the p.r. functions is unbounded universal and existential quantification. That is, we cannot define a function $f(x, y) \stackrel{\text{def}}{=} (\forall x) g(y)$. All

p.r. functions are total, and this is partly because of the lack of unbounded quantification. In fact, evaluation of p.r. functions is computable, although not necessarily practical. Thus evaluation of m.p.r. functions is also computable if we are given a transducer and trace. We will not hesitate to use unbounded quantification when describing the properties of our functions, we just cannot use unbounded quantification in the function definitions. We have not found this lack to be a serious impediment to our work — we are using the functions to describe finite state machines, after all.

Readers interested in the development of number theory from the p.r. functions should consult Peters well-known work, or a more modern, treatment such as that found in Hinmann [26], Smorynski [57], or Lewis and Papadimitrou [37]. An intriguing, although not simple, alternative approach to the p.r. functionals has been recently developed by Simmons [53].

2.4 A formal definition of the modal p.r. functions.

The class of m.p.r. functions includes a small set of initial functions, and all those functions definable from the initial functions with a finite number of applications of an even smaller set of function construction rules. The m.p.r. functions form an extension to the class of primitive recursive functions. Because the initial primitive recursive functions are also initial m.p.r. functions, and because both of the primitive recursive rules for defining new functions are also m.p.r. function definition rules, everything expressible in primitive recursive arithmetic is also expressible in m.p.r. arithmetic.

Definition 2.17: The class of m.p.r. functions
The class MPR of the modal primitive recursive functions consists of all the functions which can be generated after a finite number of steps using the following rules.

1. **Zero:** $f(\vec{x}) \stackrel{def}{=} 0$,
2. **Successor:** $f(x) \stackrel{def}{=} x + 1$,
3. **Projection:** $f(y, \vec{x}) \stackrel{def}{=} \vec{x}_y$,
4. **Substitution:** $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}), \ldots g_n(\vec{x}))$,
5. **Primitive Recursion:** $f(0, \vec{x}) = g(\vec{x}), f(r+1, \vec{x}) \stackrel{def}{=} h(r, \vec{x}, f(r, \vec{x}))$,
6. **Alphabet:** $f(\,) \stackrel{def}{=} \mathsf{Alphabet}$,
7. **Output function:** $f(x) \stackrel{def}{=} \mathsf{Out}(x)$,
8. **Component names:** $f(\,) \stackrel{def}{=} \mathsf{Components}$,
9. **Pumping number:** $f(\,) \stackrel{def}{=} \mathsf{Pump\_number}$,
10. **Enabling:** $f(x) \stackrel{def}{=} \mathit{enable}(x)$,
11. **Precedence:** $f(x, y, x', y') \stackrel{def}{=} \mathit{precedes}(x, y, x', y')$,
12. **Feedback:** $f(x, y) \stackrel{def}{=} \mathit{f\_effect}(x, y)$,
13. **Path offset:** $f(y, \vec{x}) \stackrel{def}{=} (\mathit{after}\, y)g(\vec{x})$,
14. **Component selection** $f(y, \vec{x}) \stackrel{def}{=} (\mathit{in}\, y)g(\vec{x})$.

Note that every p.r. function is also an m.p.r. functions. The defining equations for p.r. functions are quite informative: if f is p.r. then we can reduce an expression $f(n)$ to a unique integer k by a finite number of rewritings using the defining equations. The defining equations for functions which are m.p.r., but not p.r. are not so informative. We remedy this situation by developing a map $\rho : \mathsf{MPR} \to \mathsf{PR}$, which reduces each m.p.r. function to a primitive recursive function that exposes the implicit dependencies on state machines and traces. We will *define* the value of an m.p.r. expression $f(n)$ in the context of P and w, to be $(\rho f)(P, w, n)$. This interpretation is the subject of the next section.

2.5 Interpretation

One of the advantages of basing our language of functions on the p.r. recursive functions is that we can describe mappings on MPR iteratively. Every m.p.r. function is either an initial function, or defined from simpler m.p.r. functions by substitution,

primitive recursion, transition offset, or component selection. Thus, we can define ρ by first defining $\rho : \mathsf{InitialFunctions} \rightarrow \mathsf{PR}$, and then, supposing that ρ is defined on all MPR functions constructed using n or fewer applications of the defining rules, define (ρf) for a function constructed using one of the composing rules.

Definition 2.18: The value of an m.p.r. function
The *value* of a m.p.r. function f in the context (P, w) of an automaton P and trace w is defined to be $(\rho f)(P, w, \vec{x})$.

Definition 2.19: The evaluation functional, ρ
Let $P = (\langle A, O, S, start, \lambda, \delta, (P_1, \ldots P_r), \Phi \rangle$

If $f(x) \stackrel{def}{=} 0$

$(\rho\ f)(P, w, x) \stackrel{def}{=} 0$

If $f(x) \stackrel{def}{=} x + 1$

$(\rho\ f)(P, w, x) \stackrel{def}{=} x + 1$

If $f(y, \vec{x}) \stackrel{def}{=} \vec{x}_y$

$(\rho\ f)(P, w, y, \vec{x}) \stackrel{def}{=} \vec{x}_y$

If $f() \stackrel{def}{=}$ **Alphabet**

$(\rho\ f)(P, w) \stackrel{def}{=} A$

If $f() \stackrel{def}{=}$ **Components**

$(\rho\ f)(P, w) \stackrel{def}{=} \{1 \ldots, r\}$

If $f() \stackrel{def}{=}$ **Pump_number**

$(\rho\ f)(P, w, x) \stackrel{def}{=} size(S) * \prod_i size(PS(P.i))$

If $f(x) \stackrel{def}{=}$ **Out**(x)

$(\rho\ f)(P, w, x) \stackrel{def}{=} (\lambda(\Delta(P, w))_x$

If $f(x) \stackrel{def}{=}$ **enable**(x)

$$(\rho\ f)(P, w, x) \stackrel{def}{=} \begin{cases} 1 & \textbf{if } w\langle x \rangle \in \mathcal{L}(P) \\ 0 & \textbf{otherwise.} \end{cases}$$

If $f(x) \stackrel{def}{=}$ **precedes**(x, y, x', y')

$(\rho\ f)(P, w, x, y, x', y') \stackrel{def}{=}$ **RelOrder**(w, x, y, x', x')

If $f(x) \stackrel{def}{=}$ **f_effect**(x, y)

$$(\rho\ f)(P, w, x, y,) \stackrel{def}{=} \begin{cases} \phi_y(\Delta(P, w), x) & \textbf{if } 0 < x \leqslant r; \\ \langle\rangle & \textbf{otherwise.} \end{cases}$$

If $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}), \ldots, g_n(\vec{x}))$

$(\rho f)(P, w, \vec{x}) = (\rho h)(P, w, \ldots (\rho g_i)(P, w, \vec{x}), \ldots)$

If $f(0, \vec{x}) \stackrel{def}{=} g(\vec{x}), f(r + 1, \vec{x}) \stackrel{def}{=} h(r, \vec{x}), f(r, \vec{x}))$

$(\rho f)(P, w, 0, \vec{x}) \stackrel{def}{=} (\rho g)(P, w, \vec{x})$

$(\rho f)(P, w, r + 1, \vec{x})) \stackrel{def}{=} (\rho h)(P, w, r, \vec{x}, (\rho f)(P, w, r, \vec{x}))$

If $f(y, \vec{x}) \stackrel{def}{=} (\textbf{after}\ y)g(\vec{x})$

$(\rho f)(P, w, y, \vec{x}) \stackrel{def}{=} (\rho g)(P, w \cdot y, \vec{x})$

If $f(y, \vec{x}) \stackrel{def}{=} (\textbf{in}\ y)g(\vec{x})$

$$(\rho f)(P, w, y, \vec{x}) \stackrel{def}{=} \begin{cases} (\rho g)(P_y, pfactor(P, w, y), \vec{x})) & \textbf{if } 0 < y \leqslant r; \\ 0 & \textbf{otherwise.} \end{cases}$$

Theorem 2: The functional ρ is an effectively 1-1 map from the m.p.r. functions to the p.r. functions.

Proof. Note that (ρf) is obviously a p.r. function when f is an initial m.p.r.

function. Proceding by induction we can see that (ρf) must be p.r. for all functions $f \in MPR$. The function is "effectively" 1-1 because for every p.r. function f, there is a m.p.r. function, f itself, so that $f'(\vec{x}) = (\rho f)(P, w, \vec{x}) = f(\vec{x})$. That it, the map is 1-1 modulo hiding the extra, meaningless arguments.

Corollary. 1: If f is p.r. then $(\rho f)(P, w, x) = f(x)$.

Corollary. 2: For arbitrary m.p.r. function f, $(\exists P, w, \vec{x})(\rho f)(P, w, \vec{x}) > 0$ is semi-decidable, but not decidable.
Proof: Undecidability follows from the well known undecidability of the p.r. functions [20] and the inclusion of the p.r. functions in the m.p.r. functions. Semi-decidability follows from a simple dovetailing argument; the set of possible argument vectors is enumerable, so we simply test each vector until/if we find a satisfying case.

Corollary. 3: $(\rho f)(P, w, \vec{m}) = g(n)$ is decidable in time bounded by Ackermann's function for arbitrary m.p.r. functions f and g and constants m and n.
Proof: Follows from the well known decidability of closed p.r. terms [20] and reduction using ρ.

2.6 Summary

In this chapter we have examined the intuition behind the m.p.r. arithmetic and shown that the m.p.r. functions have a well-defined syntax and semantic interpretation. The feedback product form transducers are, clearly, derived from previous work in algebraic automata theory, but take on a new significance in the context of formal specification. The m.p.r. functions provide a natural and intuitive language in which to describe these, very complex, algebraic objects. We can define a functional $\rho : MPR \rightarrow PR$ so that ρf is a p.r. function which exposes the context dependencies implicit in f. Thus, we have an algebra of context dependent integer-valued functions which reflect the dynamic behavior and structure of discrete systems represented by product form transducers.

CHAPTER 3

SPECIFICATION AND VERIFICATION

One of the advantages of finite automata is that there is a clear and intuitive correspondence between automata and physical systems. Thus, the process of translating our informal understanding of a system into a formal specification of a state machine is less error prone than it might be otherwise. M.p.r. functions can be used to specify transducers in a way which preserves this intuitive correspondence and extends it to composite and real-time systems. Detailed m.p.r. specifications of quite complex systems can be made to be very compact and intuitively transparent. Specifications can be tested for accuracy and correctness using *deepening*, *abstraction*, and *construction*. We *deepen* a specification by specifying the mechanics of the system in more detail. For example, we may deepen a specification of a network by specifying the nature of the connection, the possible transmission errors, or the real-time constraints on the sites. An *abstraction* of a specification is a correctness property that follows from the original specification. The key here is to prove that the abstract correctness property (which captures some desired system behavior) is guaranteed by the original specification. Conversely, we might want to prove that a deepened specification implies the correctness of the original specification. *Construction* involves proving that a specification can be *satisfied* by at least one transducer. That is, given a specification $\mathrm{Spec}(x)$, we ask if there is some P so that $\mathrm{Spec}(x) > 0$ in the context of (P, w), for each $w \in \mathcal{L}(P)$ — is there a P so that $\mathrm{Spec}(x)$ true under every reachable state of P. A construction of a specification is a specification which implies the correctness of the original specification, and which can be shown to be satisfiable by at least one transducer. If we can find a construction of $\mathrm{Spec}(x)$, then we know that, in principle, we can build a device which implements $\mathrm{Spec}(x)$. While there can never be a *proof* that an abstract object (a specification) is an accurate representation of a physical object (a computational device), we can be reasonably confident of a specification which: corresponds to our intuitive understanding of the system, provably satisfies a variety of correctness criteria, and is finitely realizable.

The discussion of the previous paragraph can be made more precise by more precisely defining what *satisfaction* means. We say $f(\vec{x})$ is a m.p.r. *term* iff f is a m.p.r. function, and $\vec{x}$ is a list of variables. Every m.p.r. term $f(\vec{x})$ is a specification of the transducers which satisfy $f(\vec{x})$.

Definition 3.1: Satisfaction of a term by a transducer.

A transducer P *satisfies* a term $f(\vec{x})$

$$(\forall w \in \mathcal{L}(P))(\forall \vec{x}) \quad \left[\boxed{\boxed{P}_{w}} \quad f(\vec{x}) \right] > 0$$

If $f(\vec{x})$ is a refinement of $g(\vec{x})$ then every transducer which satisfies $f(\vec{x})$ should also satisfy $g(m)$. If $f(\vec{x})$ is an abstraction of $g(m)$, then every transducer which satisfies $g(\vec{x})$ should also satisfy $f(\vec{x})$. We say $f(\vec{x})$ is feasible, iff there is at least one transducer which satisfies $f(\vec{x})$.

Remark on constructions. If a specification Spec(x) describes a system which cannot be implemented by a finite state transducer, then we consider Spec(x) to be incorrect. The m.p.r. arithmetic is intended as a vehicle for studying discrete engineering, not the mathematical subject of effective computation. At present, all discrete computing devices are finite state, and thus all feasible algorithms and designs can be implemented by a finite state transducer. It is, of course, possible that someone will develop an infinite state computing device. M.p.r. arithmetic, in its present form, will not be the proper framework for describing computation on such a device. But for systems that will be implemented on digital hardware, finite transducers are a sufficiently general model. It can be argued that the convention of treating programs and systems as if they were of unbounded state: e.g, storage cells that can store arbitrary integers, simply discards "implementation details". Along similar lines, it is often claimed that finite state methodologies will obscure the essence of an algorithm or design with useless clutter. The first argument has no merit for systems and architectural level computing, where bounds on time and resources are central correctness issues. The second argument appears to result from a confusion between notation and semantics. It is true that specification of systems in terms of regular expressions or finite state diagrams requires a tedious enumeration of state sets and results in brittle specifications which are not easily extended. These are notational limitations, however, and we show in this chapter that these limitations can be overcome.

The first section of this chapter introduces various techniques for treating m.p.r. functions as elements of a sort of programming language for transducers. Unlike

traditional programs, however, these m.p.r. programs do not start with a fixed notion of control flow. Instead specifications are written to define under what conditions a transition can become enabled, how future transitions will change current state properties, and how the outputs of components will correspond to each other and global state. The second section introduces m.p.r. proof techniques. To show that a specification S implements a property Q, we show that $S \rightarrow Q$ is a *valid* expression. That is, if S is true of in the context (P, w), then Q must also be true in the context of (P, w). Clearly, it would be foolhardy to attempt to prove the validity of $S \rightarrow Q$ by inspecting all contexts. Instead we prove an expression is valid by using general properties of contexts and m.p.r. functions. The final section addresses proofs of feasibility. Certain types of m.p.r. functions called *exact grammars* are defined in such a way so that there is an algorithm for constructing a transducer from the grammar. If P is constructed via this algorithm from a grammar $\mathcal{G}$, then $\mathcal{G}$ is *satisfied* by P — $\mathcal{G}$ has non-zero value under (P, w) for every trace w of P. And if $\mathcal{G}$ is satisfied by another transducer P', then P and P' will have identical input languages, and identical outputs on every trace.

3.1 Specification Style

In this section we introduce some techniques for developing specifications at varying degree of detail. We begin with a type of function definition called *modal recursion* which allows description of state change in a very "high-level" manner. We then switch to a mode detailed, style called a *modal grammar.*

3.1.1 Modal recursion

Primitive recursive definitions of functions involve defining the value of the function on 0, and then defining the value of the function on x+1 in terms of its values on $y \leqslant x$. There is an analogous technique which is of great utility in defining modal functions. This technique involves defining the function value in the initial state, i.e., when the current trace is $\langle\rangle$ the empty trace, and then defining the value of the function after each possible transition in terms of its current value. Recall from section 2.1.1 that Initial is a boolean function which is true iff the current trace is $\langle\rangle$, the empty trace. Thus, if we write $\mathsf{Initial} \rightarrow E$, we are asserting that E must be true in the initial state. In definition 2.4 we defined $\Box f$ to be a boolean function so that $\Box f(x) = 1$ iff $(\forall u)(\mathsf{enable}(u) \rightarrow (\mathsf{after}\, u)f(x) > 0)$. Thus, if we write $\Box f(x)$, we are asserting that f(x) is non-zero in the current state (where $u = \langle\rangle$), and in all reachable future

states. We can combine these two types of assertions to describe modal functions using modal recursion.

Definition 3.2: Modal recursion.

A function $f(\vec{x})$ is modal recursive in g and h iff :

$$\text{Initial} \rightarrow f(\vec{x}) = g(\vec{x})$$
$$\Box(\text{after } a)f(\vec{x}) = h(a, \vec{x}, f(\vec{x}))$$

Note that not all modal recursive functions are m.p.r. functions. For example, $\Box(\text{after } a)\text{counter}(x) = \text{counter}(x) + 1$ is not m.p.r. [1] The main use of functions defined using modal recursion is to constrain the behavior of m.p.r functions. That is, we use modal recursive functions to describe the desired behavior of a m.p.r. function that we then describe more algorithmically.

Modal recursion is especially useful when we want to define the result of a state transition, without defining the mechanism. For example, we can define fifo and lifo queues quite simply, using this technique and the sequence functions defined in figure 2.1. Recall that $\text{rtrunc}(x::y) = x$ and $\text{ltrunc}(y:x) = x$, while $\text{rtrunc}(\langle\rangle) = \text{ltrunc}(\langle\rangle) = \langle\rangle$.

$$\text{fifo\_q} : \emptyset \rightarrow \text{Sequences}$$
$$\text{Initial} \rightarrow \text{fifo\_q} = \langle\rangle$$
$$\Box(\text{after } a)\text{fifo\_q} = \begin{cases} \langle x\rangle\text{fifo\_q} & \textbf{if } a = \text{push.x;} \\ \text{rtrunc(fifo\_q)} & \textbf{if } a = \text{pop;} \\ \text{fifo\_q} & \textbf{otherwise.} \end{cases}$$

Figure: 3.1 A fifo queue

Note that the definition of the fifo_q function is much like a standard abstract data type definition [1]. The key to the definition is that $(\text{after push.x})\text{fifo\_q}$ is obtained by pre-pending x to the current value of fifo_q. We can easily derive a lifo_q specification from the fifo_q specification by replacing rtrunc with ltrunc.

[1] Intuitively, *counter* should not be an m.p.r. function because finite state machines cannot implement unbounded counters. For proof, note that for every m.p.r. initial function f, there is a p.r. function g so that $(\text{after } u)f(x) \leq g(x)$. It is easy to show, by induction, that this holds for all m.p.r. functions. But, there is no such bound on *counter*.

$$
\begin{aligned}
&\text{lifo\_q} : \emptyset \rightarrow \text{Sequences} \\
&\text{Initial} \rightarrow \text{lifo\_q} = \langle\rangle \\
&\Box(\text{after } a)\text{lifo\_q} = \begin{cases} \langle x\rangle\text{lifo\_q} & \textbf{if } a = \text{push.x;} \\ \text{ltrunc(lifo\_q)} & \textbf{if } a = \text{pop;} \\ \text{lifo\_q} & \textbf{otherwise.} \end{cases}
\end{aligned}
$$

Figure: 3.2 A lifo queue

3.1.2 Modal grammars: Specifications of systems

For specifications that are more detailed than modal recursive functions, we adopt a style of function called a *modal grammar.* A modal grammar is a function defined as the conjunction of clauses which constrain the alphabet, components, feedback, output functions, and enabling rules of a transducer. In outline a modal grammar has the following form:

$$
\begin{aligned}
&\text{grammar}(\vec{x}) \stackrel{\text{def}}{=} \\
&\{ \\
&\quad \textbf{Alphabet} = f_{\text{alphabet}}(\vec{x}) && (1) \\
&\quad \bigwedge \textbf{Outputs} = f_{\text{outputs}}(\vec{x}) && (2) \\
&\quad \bigwedge \textbf{Components} = f_{\text{components}}(\vec{x}) && (3) \\
&\quad \bigwedge (\forall i \in \textbf{Outputs})\, \textbf{Out}(i) = f_{\text{output}}(i, \vec{x}) && (4) \\
&\quad \bigwedge (\forall c \in \textbf{Components})(\textbf{in } c)\Box \text{Spec}_c(\vec{x}) && (5) \\
&\quad \bigwedge (\forall a \in \textbf{Alphabet})\, \textbf{enable}(a) = f_{\text{enable}}(a, \vec{x}) && (6) \\
&\}
\end{aligned}
$$

Figure: 3.3 A modal grammar

Line (1) defines the alphabet of the transducer. Line (2) defines the outputs. Recall that the output of a transducer is given by a single function Out so that Out(x) is the x^{th} element of the output tuple of the transducer. It is often more clear to define a set of output function names Outputs = $\{n_0, ..., n_k\}$ and let n_i = Out(i). Thus, lines (2) and (4) define Out and define the pseudonyms for each element of the output tuple. Line (3) defines the component names, and line (5) associates each component with a *type* — a specification which is satisfied in the component. When we write (in c)$\Box$Spec$_c(\vec{x})$, we are asserting that every enabled path within component c must reach a state where Spec$_c(\vec{x})$ is true. The final line, line (6) defines the enabling rules of the transducer. Clearly if $f_{\text{alphabet}}(\vec{x})$,$f_{\text{outputs}}(\vec{x})$, $f_{\text{components}}(\vec{x})$, $f_{\text{enable}}(\vec{x}$, and every Spec$_c$ are m.p.r. functions, then grammar$(\vec{x})$ must also be an m.p.r. function.

There is no guarantee that $grammar(\vec{x})$ is satisfied by any transducers at all. For example, a grammar containing the clause $enable(a) = \neg(enable(a))$ evaluates to 0 under all contexts. Similarly, a grammar which allows $f_{enable}(a, \vec{x}) \bigwedge \neg(in\ c)\ enable(f\_effect(a, c))$ for some $c \in Components$ is not satisfied by any transducer. We have defined $\mathcal{L}(P)$ so that $w \in \mathcal{L}(P)$ only if the paths induced by w for each component are in the trace languages of the components. $enable(a)$ is true in context (P, w) only if $pfactor(w::a, c) \in \mathcal{L}(P_c))$ — only if $(in\ c)\ enable(f\_effect(c, a))$ is true under (P, w) for each factor c. This is the reason that we write $(in\ c)\Box Spec_c(\vec{x})$ *in* line (5), instead of the weaker $\Box(in\ c)Spec_c(\vec{x})$. The first assertion states that if w is enabled within c, then w must reach a state where $Spec_c(\vec{x})$ is true. The second assertion states that if w is enabled in the composite system, and thus $f\_effect(w, c)$ is enabled in the component, then $(after\ w)(in\ c)Spec_c(\vec{x})$ must be true. Since $f\_effect(w, c)$ may contain more than one element, the first assertion implies the second assertion, but the reverse implication is not necessarily correct.

When we write specifications, we will depart from the basic outline of a grammar whenever it seems convenient. Grammars provide a convenient style of certain kinds of specifications. But the order of the clauses and the exact form of the grammar is not critical. It is also possible to omit some clauses, or to add additional clauses, or to replace the equalities with weaker clauses. A modal grammar without lines (3) and (5) specifies a system with no components. A modal grammar with implication clauses in place of equalities allows for non-determinism. For example, figure 3.4 is a grammar, which uses the implication $Condition(x) \rightarrow Data = x$ to assert that if the condition holds, the output is fixed. If the condition is not true, then we make no claims about the output. This style is especially useful for real-time systems and circuits, where we know that a condition will be true within at least k ticks, but need to allow for some uncertainty.

3.1.3 An example.

We can define a specification of a storage cell and then use the cell as a component of a memory bank and a fifo. To make the specification a little more interesting, we specify a *clocked* memory cell.

Notation. We use a dot index notation in preference to subscripting whenever the dot index notation seems clearer. Thus, we write $\{load.0, ..., load.(n-1)\}$ in preference to, the equivalent $\{load_0, ..., load_{n-1}\}$. This notation is especially useful when we have multiple indexes, e.g. $x.(y_1, ..., y_n)$ or indexes that are functional expressions, e.g., $x.f(y)$.

$$
\begin{array}{l}
\mathrm{cell}(\mathrm{limit}, \mathrm{latch\_time}, \mathrm{delay},) \stackrel{\mathrm{def}}{=} \\
\{ \\
\quad \mathbf{Alphabet} = \{\mathrm{clk}, \mathrm{load}.x : 0 \leqslant x < \mathrm{limit}\} \\
\quad \wedge\, \mathbf{Outputs} = \{\mathrm{Data}\} \\
\quad \wedge\ (\forall x \neq x')\, \mathbf{precedes}(\mathrm{load}.x', 0, \mathrm{load}.x) \\
\qquad \wedge\, \mathbf{precedes}(\mathrm{load}.x, 1, \mathrm{clk}, \mathrm{delay}) \rightarrow \mathrm{Data} = x \\
\quad \wedge\, \mathbf{enable}(\mathrm{clk}) = 1 \\
\quad \wedge\, \mathbf{enable}(\mathrm{load}.x) = (\forall x)\, \mathbf{precedes}(\mathrm{load}.x, 1, \mathrm{clk}, \mathrm{latch\_time}) \\
\}
\end{array}
$$

Figure: 3.4 A clocked storage cell

We can now specify a memory bank made up of cells. A key feature of a memory bank is that the output of the bank does not make the contents of all cells available simultaneously. In order for the user of a bank to see the contents of cell y, the user must instruct the bank to read.y, and make the contents of cell y visible.

$$
\begin{array}{l}
\mathrm{bank}(\mathrm{ncells}, \mathrm{limit}, \mathrm{latch\_time}, \mathrm{delay},) \stackrel{\mathrm{def}}{=} \\
\{ \\
\quad \mathbf{Components} = \{\mathrm{mcell}.0, \ldots, \mathrm{mcell}.(\mathrm{ncells} - 1)\} \\
\quad \wedge (\forall 0 \leqslant y < \mathrm{ncells})(\mathbf{in}\ \mathrm{mcell}.y) \Box \mathrm{cell}(\mathrm{limit}, \mathrm{latch\_time}, \mathrm{delay}) \\
\quad \wedge \mathbf{Alphabet} = \{\mathrm{clk}, \\
\qquad\qquad \mathrm{read}.x, \mathrm{load}.(x, y) : 0 \leqslant x < \mathrm{limit}, 0 \leqslant y < \mathrm{ncells}\} \\
\quad \wedge\, \mathbf{f\_effect}(\mathrm{write}.(x, y), y') = \begin{cases} \langle \mathrm{load}.x \rangle & \textbf{if } y = y'; \\ \langle\rangle & \textbf{otherwise.} \end{cases} \\
\quad \wedge\, \mathbf{f\_effect}(\mathrm{read}.x) = \langle\rangle \\
\quad \wedge\, \mathbf{f\_effect}(\mathrm{clk}, y) = \langle \mathrm{clk} \rangle \\
\quad \wedge\, \mathbf{Outputs} = \{\mathrm{Data}, \mathrm{Ready}\} \\
\quad \wedge\, \mathrm{Ready} = (\forall y)\, \mathbf{precedes}(\mathrm{read}.y, 1, \mathrm{clk}, \mathrm{delay} + \mathrm{latch\_time}) \\
\qquad\qquad \wedge (\forall y, x)\, \mathbf{precedes}(\mathrm{write}.(y, x), 1, \mathrm{clk}, \mathrm{delay} + \mathrm{latch\_time}) \\
\quad \wedge\, \mathrm{Ready} \wedge\ (\forall y' \neq y)\, \mathbf{precedes}(\mathrm{read}.y', 1, \mathrm{read}.y, 1) \\
\qquad\qquad \rightarrow \mathrm{Data} = (\mathbf{in}\ \mathrm{mcell}.y)\mathrm{Data} \\
\quad \wedge\, \mathbf{enable}(\mathrm{clk}) = 1 \\
\quad \wedge\, \mathbf{enable}(\mathrm{read}.y) = \mathrm{Ready} \\
\}
\end{array}
$$

Figure: 3.5 A clocked memory bank

The fifo is built from $n+1$ cells, one acting as a pointer to the head of the queue. When a new item is *pushed* onto the stack, the tail queue cell loads the new item, and

in parallel, every other queue cell loads the output of its predecessor in the queue. The push operation also causes the pointer cell to advance by 1 place. A *pop* operation causes the pointer cell to point at the predecessor of the head, and leaves the other cells unchanged.

$$
\begin{array}{l}
\text{clocked\_fifo}(t_{load}, t_{select}, \text{qlength}, \text{range}) \stackrel{\text{def}}{=} \\
\{ \\
\textbf{Alphabet} = \{\text{clk}, \text{pop}, \text{push.x} : x \in X\} \\
\bigwedge \textbf{Components} = \{\text{ptr}, \text{mcell.1}, \ldots, \text{mcell.qlength}\} \\
\bigwedge \textbf{Outputs} = \{\text{Ready}, \text{Head}, \text{Empty}, \text{Full}\} \\
\bigwedge (\forall i)(\textbf{in}\ \text{mcell.i})\Box\text{cell}(\text{limit}, t_{load}, t_{select}) \\
\bigwedge (\textbf{in}\ \text{ptr})\Box\text{cell}(\text{qlength}+1, t_{load}, t_{select}) \\
\bigwedge \text{headptr} = (\textbf{in}\ \text{ptr})\text{Data} \\
\bigwedge \text{Head} = (\textbf{in}\ \text{mcell.headptr})\text{Data} \\
\bigwedge \text{Empty} = (\text{headptr} = 0) \\
\bigwedge \text{Full} = (\text{headptr} = \text{qlength}) \\
\bigwedge \text{Ready} = \left(\begin{array}{l} \textbf{precedes}(\text{push.x}, 1, \text{clk}, t_{load} + t_{select}) \\ \bigwedge \textbf{precedes}(\text{pop}, 1, \text{clk}, t_{load} + t_{select}) \end{array} \right) \\
\bigwedge \text{Mdata}(i) = (\textbf{in}\ \text{mcell.i})\text{Data} \\
\bigwedge \text{f\_effect}(a, \text{mcell.i}) = \left\{ \begin{array}{ll} \langle \text{clk} \rangle & \textbf{if}\ a = \text{clk} \\ \langle \text{load.x} \rangle & \textbf{if}\ a = \text{push.x} \bigwedge i = 0 \\ \langle \text{load.Mdata}(i-1) \rangle & \textbf{if}\ a = \text{push.x} \bigwedge i > 0 \\ \langle\rangle & \textbf{otherwise.} \end{array} \right. \\
\bigwedge \text{f\_effect}(a, \text{ptr}) = \left(\begin{array}{ll} \langle \text{load.}(\textbf{in}\ \text{ptr})\text{Data} \rangle + 1 & \textbf{if}\ a = \text{push.x} \\ \langle \text{load.}(\textbf{in}\ \text{ptr})\text{Data} \rangle - 1 & \textbf{if}\ a = \text{pop} \\ \langle \text{clk} \rangle & \textbf{if}\ a = \text{clk} \end{array} \right. \\
\bigwedge \textbf{enable}(\text{push.x}) = \text{Ready} \bigwedge (\neg \text{Full}) \\
\bigwedge \textbf{enable}(\text{clk}) = 1 \\
\bigwedge \textbf{enable}(\text{pop}) = \text{Ready} \bigwedge (\neg \text{Empty}) \\
\}
\end{array}
$$

Figure: 3.6 A clocked fifo queue

It should be easy to see that clocked_fifo does implement a fifo, but to prove this formally, we will want to prove that:

$$
\begin{array}{l}
\Box\text{clocked\_fifo}(t, t', x, y) \\
\rightarrow \Box \left(\text{Ready} \rightarrow \left(\begin{array}{l} (\neg \text{Empty}) \rightarrow \text{Head} = \text{head}(\text{lifo\_q}) \\ \bigwedge \text{Empty} = (\text{lifo\_q} = \langle\rangle) \\ \bigwedge \text{Full} = (\text{length}(\text{lifo\_q}) = x) \end{array} \right) \right)
\end{array}
$$

To prove these, and other correctness properties, we need the methods developed in the next section.

3.2 Proofs

Truth and Theorems. Recall that we define a non-zero value to be *true*, and let 0 denote *false*. An expression is *valid* iff it is true in every context. For example $(x+1)$ and $(\mathsf{after}\,a)\mathsf{precedes}(b,0,a,0)$ are both valid. A *theorem* is an expression which includes function variables, and which is valid for every instantiation. For example $(f(0) = k \wedge f(x) = k \rightarrow f(x+1) = k) \rightarrow f(x) = k$ and $(\mathsf{in}\,c)(E \wedge F) \leftrightarrow (\mathsf{in}\,c)E \wedge (\mathsf{in}\,c)F$ are both theorems because the first is valid for all functions f and the second is valid for all expressions E and F.

3.2.1 Inherited algebra

We let the function $sgn(0) \stackrel{\text{def}}{=} 0$ and $sgn(x+1) = 1$ reduce an arbitrary value to a boolean value. To make arithmetic reasoning a little easier, we define the logical connectives $\wedge$, $\vee$, $\neg$ and $\rightarrow$ as boolean functions.

$$\begin{array}{l} x \wedge y \stackrel{\text{def}}{=} sgn(x * y) \\ x \vee y \stackrel{\text{def}}{=} sgn(x + y) \\ \neg x \stackrel{\text{def}}{=} 1 \dot{-} sgn(x) \\ x \rightarrow y \stackrel{\text{def}}{=} (\neg x) \vee y \end{array} \quad (0)$$

We also define the arithmetic relations, "=", "<" and ">" as boolean functions. Note that $x = y$ is a boolean expression with value 1 if the sum of the differences, $(x \dot{-} y) + (y \dot{-} x)$, is zero.

$$\begin{array}{l} x = y \stackrel{\text{def}}{=} \neg((x \dot{-} y) + (y \dot{-} x)) \\ x < y \stackrel{\text{def}}{=} sgn(y \dot{-} x) \\ x > y \stackrel{\text{def}}{=} sgn(x \dot{-} y) \\ x \leqslant y \stackrel{\text{def}}{=} sgn((y+1) \dot{-} x) \\ x \geqslant y \stackrel{\text{def}}{=} sgn((x+1) \dot{-} y) \end{array} \quad (1)$$

At this point, the task of integrating m.p.r. into the p.r. functions starts to pay off. Goodstein [20, 21] developed a collection of proof rules for terms of the primitive recursive arithmetic. The rules capture some elementary equalities in basic algebra and the induction rule. The rules are intended as fundamental starting points in a formal treatment of arithmetic, but, in keeping with our informal approach, we will take these rules as some elementary theorems about m.p.r. expressions.

$$(E = F \bigwedge F = E' \rightarrow E = E')$$
$$(f(x) = g(x) \rightarrow f(n) = g(n))$$
$$(E = F \rightarrow f(E) = f(F))$$
$$(f(x) = f(x+1) \rightarrow f(x) = f(0))$$
$$f(x+1) = 1 + f(x) \rightarrow f(x) = f(0) + x)$$
$$(f(x+1) = f(x) \dot{-} 1 \rightarrow f(x) = f(0) \dot{-} x)$$
$$(f(0,\vec{x}) = g(0,\vec{x}) \bigwedge f(r+1,\vec{x}) = g(r+1,\vec{x}) \rightarrow f(y,\vec{x}) = g(y,\vec{x}))$$

Figure: 3.7 Goodstein's rules for primitive recursive arithmetic

Theorem 3: The proof rules of primitive recursive arithmetic are m.p.r. theorems.

Proof: Follows from the definition of ρ and standard algebra.

Theorem 4: If $f(\vec{x}) \stackrel{def}{=} E(\vec{x})$ is a defining equation of f, then $f(\vec{x}) = E(\vec{x})$ is a valid m.p.r. expression.

Proof: From definition of ρ.

The proof rules and valid expressions inherited from the p.r. arithmetic do not allow us to deduce much about m.p.r. expressions which are not p.r. expressions. We could simply reason about these expressions by translating them into p.r. expressions using ρ. But this would defeat our purpose. The next section develops general properties of m.p.r modal functions that are useful in many proofs.

3.2.2 Modal proofs

This section, finally, we introduce methods for proving properties of modal expressions. We begin by developing a battery of theorems that allow us to distribute after and in over expressions. We then turn to methods specific to multi-level reasoning in systems with components, and conclude with some theorems about precedes. During the course of the chapter, we introduce a number of useful derived functions.

We begin with a fundamental theorem that allows us to replace expressions in the scope of after and in with their defined meanings. That is, if $f(x) \stackrel{def}{=} E(\vec{x})$ we know that $(\text{in } c)f(\vec{x}) = (\text{in } c)E(\vec{x})$. Note that '=' is much weaker than '$\stackrel{def}{=}$'. The equality $f(x) = E(x)$ does <u>not</u> imply that $(\text{in } c)f(x) = (\text{in } c)E(\vec{x})$. For example, $\text{precedes}(a,1,b,0)$ may be true in the current context, so $\text{precedes}(a,1,b,0) = 1$, but it does not necessarily follow that $(\text{in } c\ \text{precedes}(a,1,b,0))$ is equal to $(\text{in } c)1$.

M.p.r. theorem 1: If $f(\vec{x}) \stackrel{\text{def}}{=} E$ then $(\text{after}\, u)f(\vec{x}) \stackrel{\text{def}}{=} (\text{after}\, u)E$ and $(\text{in}\, c)f = (\text{in}\, c)E$.

Proof: From definition of ρ.

3.2.3 Path division and path induction

The most basic m.p.r. theorem for after allows us to divide paths into segments. This m.p.r. theorem states that evaluating an expression $f(\vec{x})$ in the state reached by following $u \cdot v$ from the current state is equivalent to following u from the current state, and then evaluating $(\text{after}\, v)f(\vec{x}$. Furthermore, following the null path, $\langle\rangle$ is shown to have no effect on function evaluation.

M.p.r. theorem 2:

$$(\text{after}\langle\rangle)f(\vec{x}) = f(\vec{x}))$$

$$(\text{after}\, x \cdot y\ f)(\vec{z}) = (\text{after}\, x\ (\text{after}\, y)f)(\vec{z}))$$

Proof: Let $f'(y, \vec{x}) \stackrel{\text{def}}{=} (\text{after}\, y)f(\vec{x})$. For the empty path property:

$$\begin{aligned}(\rho f')\ & (P, w, \langle\rangle, \vec{x}) \\ & = (\rho f)(P, w \cdot \langle\rangle, \vec{x}) \\ & = (\rho f)(P, \vec{x})\end{aligned}$$

For the "path division" property:

$$\begin{aligned}(\rho f')\ & (P, w, x \cdot y, \vec{z}) \\ & = (\rho f)(P, w \cdot x \cdot y, \vec{z}) \\ & = (\rho f')(P, w \cdot x, y, \vec{x})\end{aligned}$$

We now prove that we can prove properties by induction on enabled sequences. Suppose that $f(x)$ is true in the current state. Suppose further, that whenever u leads to a state where $f(x)$ is true, then every enabled sequence $u :: a$ must also lead to a state where $f(x)$ is true. It should follow that $f(x)$ is true in every reachable state.

M.p.r. theorem 3:

$$\begin{aligned}& (\text{after}\langle\rangle)f(x) \\ & \bigwedge(\forall u)\left(\begin{array}{l}(\text{enable}(u) \bigwedge (\text{after}\, u)f(x)) \\ \quad \rightarrow (\forall a)(\text{enable}(u :: a) \rightarrow (\text{after}\, a :: a)f(x)\end{array}\right) \\ & \rightarrow (\forall u)(\text{enable}(u) \rightarrow (\text{after}\, u)f(x)\end{aligned}$$

Proof: Suppose that the premise is true and consider some sequence u so that $\text{enable}(u)$. If $u = \langle\rangle$ the conclusion is immediate from the premise. Now suppose

the theorem holds for all u with length bounded by k. Let $u = w :: b$ and let the length of w be k. It follows, by the induction hypothesis, that $(\mathbf{after}\, w)f(x)$, and by the conclusion, that $(\mathbf{after}\, w)f(x) \rightarrow (\mathbf{after}\, wb)f(x)$. Thus, we have shown $(\mathbf{after}\, wb)f(x)$.

3.2.4 Distributive m.p.r. theorems

Expressions An *expression* is simply an unnamed function. For example, $x + 5$ abbreviates $f(x)$ where $f(x) \stackrel{\text{def}}{=} x + 5$. In m.p.r. arithmetic, expressions may also involve modal functionals, and this requires us to be a little careful. For example, the meaning of $(\mathbf{in}(\mathbf{after}\, u)f(x))g(h(y))$ may not be immediately clear. The rules for evaluation of m.p.r. expressions, however, are unambiguous, and can be used to clarify even the most complex expression. First, we note that for any expression $f(....)$ the arguments must be evaluated before the function f can be evaluated. So $(\mathbf{in}(\mathbf{after}\, u)f(x)\ g)(h(y))$, means "apply $(\mathbf{in}(\mathbf{after}\, u)f(x))g$ to the result of evaluating $h(y)$ in the current context." Fix k so that $h(g(x)) = k$. Then we note that we have an expression of the form $(\mathbf{in}\, f'(x))g(k)$, where $f'(x) \stackrel{\text{def}}{=} (\mathbf{after}\, u)f(x)$. For the evaluation to continue, we need to evaluate $f'(x)$ in the current context. Pick c so that $c = f'(x)$. We now have $(\mathbf{in}\, c)g(k)$, which is comprehensible. The original expression is revealed to mean, " evaluate $g(y)$, then evaluate $(\mathbf{after}\, u)f(x)$, use the second result to determine a new context, and evaluate g on the first result in the new context. To make expressions a little more concrete consider some examples. We write $(\mathbf{in}\, c)(E \bigwedge F)$ to force evaluation of the entire expression $E \bigwedge F$ within the context of component c. Such an expression is an abbreviation of $f(x)$ where $f \stackrel{\text{def}}{=} (\mathbf{in}\, x)f$ and $f' \stackrel{\text{def}}{=} (E \bigwedge F)$. Similarly, we write $(\mathbf{after}\, a)(E \bigwedge F)$ to force the evaluation of the expression $E \bigwedge F$ in the state reached via an a transition.

In general, an expression of the form $(\mathbf{in}\, E)E'(F)$ where E,E', and F are expressions, is evaluated by reducing E and F to constant values in the current context, and then applying E' to the constant value obtained from F, in the context of the factor named by the value of E. Similarly, an expression of the form $(\mathbf{after}\, E)E'(F)$ where E,E', and F are expressions, is evaluated by reducing E and F to constant values in the current context, and then applying E' to the constant value obtained from F, in the context of the future state reached by following the value of E.

In order to be able to investigate modal expressions, we need to be able to *distribute* the modal modifiers over the constituent parts of the expressions.

M.p.r. theorem 4: Distributive laws.

$$(\mathbf{after}\, x\ f(E_1, ..., E_n)) = (\mathbf{after}\, x) f((\mathbf{after}\, x\ E_1), ..., (\mathbf{after}\, x) E_n)$$

and

$$(\mathbf{in}\, x\ f(E_1, ..., E_n)) = (\mathbf{in}\, x) f((\mathbf{in}\, x\ E_1), ..., (\mathbf{in}\, x) E_n$$

Proof: Follows from the definition of ρ.

A useful consequence of this result is that primitive recursive functions are state independent.

M.p.r. theorem 5: Distributive law for non-modal functions.
If $(f \in PR)$

$$((\mathbf{after}\, y) f(\vec{x}) = f(\vec{x}) = (\mathbf{in}\, y) f(\vec{x})).$$

Suppose that we know that $(\mathbf{in}\, c)(E \rightarrow F)$. For clarity of notation, we write implication in functional form to get: $(\mathbf{in}\, c)(\mathrm{implies}(E, F))$. By the distributive law we get: $(\mathbf{in}\, c\ \mathrm{implies})((\mathbf{in}\, c)E, (\mathbf{in}\, c)F)$. Since implication is state independent, we can drop the first modifier to get: $\mathrm{implies}((\mathbf{in}\, c\ E,), (\mathbf{in}\, c)F)$. We can rewrite the expression in infix form to conclude: $(\mathbf{in}\, c)E \rightarrow (\mathbf{in}\, c)F$.

Rules which allow us to move expressions used as modal parameters out of enclosing expressions are grouped together as a single m.p.r. theorem, theorem 6. The first rule says that an expression $(\mathbf{after}\, y\ (\mathbf{after}\, E) f)(\vec{x})$ can be transformed to an equivalent expression $(\mathbf{after}\, y \cdot (\mathbf{after}\, y) E\ f)(\vec{x})$. That is, is we evaluate $(\mathbf{after}\, y) E$ to get some path ν, we can rewrite $(\mathbf{after}\, y\ (\mathbf{after}\, E) f)(\vec{x})$ as an equivalent expression $(\mathbf{after}\, y \nu) f(\vec{x})$. The other rules are similar.

M.p.r. theorem 6: Modal parameter distribution.

$$(\mathbf{after}\, y\ ((\mathbf{after}\, E) f))(\vec{x}) = (\mathbf{after}\, y \cdot (\mathbf{after}\, y) E\ f)(\vec{x}).$$
$$(\mathbf{after}\, y\ ((\mathbf{in}\, E\ f))(\vec{x}) = (\mathbf{after}\, y\ (\mathbf{in}(\mathbf{after}\, y) E) f(\vec{x}).$$
$$(\mathbf{in}\, y\ (\mathbf{in}\, E\ f))(\vec{x}) = (\mathbf{in}(\mathbf{in}\, y) E(\vec{x}).$$
$$(\mathbf{in}\, y\ ((\mathbf{after}\, E) f))(\vec{x}) = (\mathbf{in}\, y\ (\mathbf{after}(\mathbf{in}\, y) E) f(\vec{x}).$$

A consequence of these first m.p.r. theorems is that many primitive recursive functionals can be factored out of expressions. This facilitates algebraic analysis of modal expressions. Several important examples of this factoring are are given in the next theorem.

M.p.r. theorem 7:

$$(\mathbf{after}\,x\ \textstyle\sum_{y=0}^{i} f)(y,\vec{z}) = \sum_{y=0}^{i}(\mathbf{after}\,x)f(y,\vec{z}).$$

$$(\mathbf{in}\,x\ \textstyle\sum_{y=0}^{i} f)(y,\vec{z}) = \sum_{y=0}^{i}(\mathbf{in}\,x)f(y,\vec{z}).$$

$$(\mathbf{after}\,x\ (\forall y < k)f)(y,\vec{z}) = (\forall y < k)(\mathbf{after}\,x)f(y,\vec{z}).$$

$$(\mathbf{in}\,x\ (\forall y < k)f)(y,\vec{z}) = (\forall y < k)(\mathbf{in}\,x)f(y,\vec{z}).$$

$$(\mathbf{after}\,x\ (\exists y < k)f)(y,\vec{z}) = (\exists y < k)(\mathbf{after}\,x)f(y,\vec{z}).$$

$$(\mathbf{in}\,x\ (\exists y < k)f)(y,\vec{z}) = (\exists y < k)(\mathbf{in}\,x)f(y,\vec{z}).$$

Proof: (We just prove the first equation, as the others are nearly identical).

$$\begin{aligned}
&(\mathbf{after}\,x)\ (\textstyle\sum_{y=0}^{0} f)(y,\vec{x}) \\
&\quad = \{\ \textit{By theorem 1}\}(\mathbf{after}\,x)f(0,\vec{x}) \\
&\quad = \{\ \textit{By def of}\ \textstyle\sum\ \}\ \textstyle\sum_{y=0}^{0}(\mathbf{after}\,x)f(0,\vec{x}) \\
&(\mathbf{after}\,x)\ (\textstyle\sum_{y=0}^{i+1} f)(y,\vec{x}) \\
&\quad = \{\ \textit{By theorem 1}\} \\
&\quad (\mathbf{after}\,x)(f(j,\vec{x}) + \textstyle\sum_{y=0}^{i} f(0,\vec{x})) \\
&\quad = \{\ \textit{By theorem 5}\} \\
&\quad (\mathbf{after}\,x\ (f(j+1,\vec{x})) + (\mathbf{after}\,x)\textstyle\sum_{y=0}^{i} f(y,\vec{x}) \\
&\quad = \{\ \textit{By the induction hypothesis}\ \} \\
&\quad (\mathbf{after}\,x\ (f(j+1,\vec{x})) + \textstyle\sum_{y=0}^{i}(\mathbf{after}\,x)f(y,\vec{x}) \\
&\quad = \{\ \textit{By definition of the}\ \textstyle\sum\ \textit{functional}\ \} \\
&\quad \textstyle\sum_{y=0}^{i+1}(\mathbf{after}\,x)f(y,\vec{x})
\end{aligned}$$

3.2.5 Reasoning about components

The "compositional theorems" that are the staple of process based formal methods are not necessary in m.p.r. arithmetic. We note that if $E \rightarrow F$ is a m.p.r. theorem, then $(\mathbf{in}\,x)(E \rightarrow F)$ must also be a m.p.r. theorem. This follows from elementary semantical considerations. Using the distributive laws, we can also deduce $(\mathbf{in}\,x)E \rightarrow (\mathbf{in}\,x)F$.

Theorem 5: If any one of R, $(\forall c \in \mathrm{Components})(\mathbf{in}\,x)R$ or $(\mathbf{after}\,x)R$ are m.p.r. theorems, than all three are m.p.r. theorems.[2]

[2] In modal logics, this is often expressed as a *necessitation* rule, and can lead to some difficulties not encountered here.

Finally, we need a m.p.r. theorem that allows us to *invert* expressions of the form $(\text{after } u)(\text{in } c)f(\vec{x})$. The value of such an expression will reflect the value of $(\text{in } c)f(\vec{x})$ in the state reached by following u from the current state. But, the effect of u on component c is determined by $\text{f\_effect}(u, c)$. Thus, we can invert the expression to obtain an equivalent expression, $(\text{in } c)(\text{after } \text{f\_effect}(u, c))f(\vec{x})$.

M.p.r. theorem 8: Let $f'(y, \vec{x}) \stackrel{\text{def}}{=} (\text{after } y)f(\vec{x})$.

$$(\text{after } u)(\text{in } c)f(\vec{x}) = (\text{in } c)f'(\text{f\_effect}(u, c), \vec{x}).$$

The meaning of $(\text{in } c)(\text{after } \text{f\_effect}(u, c))f(\vec{x})$ is quite different from that of $(\text{in } c)(\text{after } \text{f\_effect}(u, c))f(\vec{x})$. The added parenthesis of the second expression causes $\text{f\_effect}(u, c)$ to be evaluated within component c. Thus, the c in $\text{f\_effect}(u, c)$ in the second expression refers to component c of component c — two levels down in the hierarchy.

3.2.6 Reasoning about precedes

We begin by verifying that Initial works as advertised.

Theorem 6:

$$(\rho\text{Initial})(P, w) > 0 \Leftrightarrow w = \langle\rangle.$$

Proof:

$$\begin{aligned}
(\rho\text{Initial})\ & (P, w) \\
&= (\forall x \in (\rho\,\text{Alphabet})(P, w))(\rho\text{Past})(P, w, x, 1) = 0 \\
&= (\forall x \in P.A)(\rho\,\text{precedes})(P, w, 0, 1, x, 1) = 0 \\
&= (\forall x \in P.A)\text{RelOrder}(w, 0, 1, x, 1) = 0 \\
&= (w = \langle\rangle)
\end{aligned}$$

M.p.r. theorem 9:

$$\text{Initial} \rightarrow (\text{after } u)\,\text{precedes}(x, y, x', y') = \text{RelOrder}(u, x, y, x', y')$$

Proof: From definition of $(\rho\,\text{precedes})$.

3.3 Exact modal grammars

The most unconventional aspect of our proof theory is the concept of syntactic proofs of finite state realizability. We will show that there are certain syntactic restrictions

such that if $\mathcal{G}(x)$ is a grammar that meets the restrictions, then for each m, $\mathcal{G}(m)$ is satisfied by exactly one (minimal) product form automaton. Thus, the construction of $\mathcal{G}$ is a proof that the system specified by $\mathcal{G}(m)$ is realizable as a finite state machine. Functions of this restricted form are called *exact modal grammars.*

3.3.1 Syntax of exact grammars

The basic form of an exact grammar is defined in figure 3.3 above and repeated in definition 3.5 below. A grammar is exact if it follows the grammar form exactly, and uses only a restricted subset of the m.p.r functions. A *unitary exact grammar*, which defines a flat transducer, is permitted to use only precedes to examine the current state. We do not permit the use of any of the other modal initial functions or composition rules. Thus, the value of $\mathcal{G}(m)$, when $\mathcal{G}$ is a unitary exact grammar, depends entirely on the trace. A *composite* exact grammar is given a little more latitude. The functions used in such a grammar may also be defined to depend on the output of the factors.

Definition 3.3: The class mpr(precedes).

The class mpr(precedes) is the smallest set of functions including all the p.r. initial functions, precedes, and all functions that can be defined from mpr(precedes) functions using substitution and primitive recursion.

Definition 3.4: The class $\mathrm{mpr}(\mathrm{precedes}, \mathrm{in}_0)$.

The class $\mathrm{mpr}(\mathrm{precedes}, \mathrm{in}_0)$ is the smallest set of functions including all the p.r. initial functions, precedes, and all functions that can be defined from mpr(precedes) functions using substitution, primitive recursion, and the rule $f(c,x) \stackrel{\mathrm{def}}{=} (\mathrm{in}\ c)\,\mathrm{Out}(x)$.

Definition 3.5: Exact grammars.

A function $f(\vec{x})$ is an *exact grammar* iff it is of the following form and satisfies the follwing conditions.

$$
\begin{aligned}
&\mathrm{grammar}(\vec{x}) \stackrel{\mathrm{def}}{=} \\
&\{ \\
&\quad \mathrm{Alphabet} = f_{\mathrm{alphabet}}(\vec{x}) && (1)\\
&\quad \bigwedge \mathrm{Outputs} = f_{\mathrm{outputs}}(\vec{x}) && (2)\\
&\quad \bigwedge \mathrm{Components} = f_{\mathrm{components}}(\vec{x}) && (3)\\
&\quad \bigwedge (\forall i \in \mathrm{Outputs})\,\mathrm{Out}(i) = f_{\mathrm{output}}(i, \vec{x}) && (4)\\
&\quad \bigwedge (\forall c \in \mathrm{Components})(\mathrm{in}\ c)\Box \mathrm{Spec}_c(\vec{x}) && (5)\\
&\quad \bigwedge (\forall a \in \mathrm{Alphabet})\,\mathrm{enable}(a) = f_{\mathrm{enable}}(a, \vec{x}) && (6)\\
&\}
\end{aligned}
$$

- The functions $f_{components}$, $f_{alphabet}$, and $f_{outputs}$ are all primitive recursive.
- And either:

  $f_{components}(\vec{x}) = \emptyset$ and f_{output}, and f_{enable} are functions in mpr(**precedes**).

  or $f_{components}(\vec{x}) \neq \emptyset$, and f_{output}, f_{enable}, and f_{effect} are functions in mpr(**precedes**, in_0), and each $Spec_c$ is an exact grammar.

3.3.2 The exact grammar theorem

Two transducers are *black box equivalent* iff they accept the same input sequences and generate identical output. That is, P is black box equivalent to P′ iff $\mathcal{L}(P) = \mathcal{L}(P')$ and $(\forall w \in \mathcal{L}(P))\lambda_P(w) = \lambda_{P'}(w)$. Intuitively, black box equivalent transducers represent systems that cannot be distinguished by the environment: they behave identically. We will show that if $\mathcal{G}$ is an exact grammar, then there will be at least one transducer P which satisfies $\mathcal{G}(m)$, and all other transducers which satisfy $\mathcal{G}(m)$ will be black-box equivalent to P. Note that two transducers that are isomorphic, perhaps with different labelings of the transitions, are not necessarily black box equivalent. For example, an automaton which outputs 1 when it has received an even number of a symbols and 0 otherwise, may be homomorphically equivalent [17, 28] to an automaton which outputs 1 when it has received an odd number of a's and 0 otherwise. These transducers do not, however, model equivalent devices. Of course, we can add a filter to one of the transducers, to make it black box equivalent.

Definition 3.6: Black-box equivalence.

Transducers P and P′ are *black box equivalent* iff $\mathcal{L}(P) = \mathcal{L}(P')$ and $(\forall w \in \mathcal{L}(P))\lambda_P(\Delta(P, w)) = \lambda_{P'}\Delta(P', w))$.

Lemma 1: The transducers that satisfy an exact grammar must be black-box equivalent.

Proof. Suppose both P and P′ satisfy an exact grammar $\mathcal{G}$ on $\vec{m}$. By definition of $\mathcal{L}$, the empty string must belong to both $\mathcal{L}(P)$ and $\mathcal{L}(P')$. Suppose, that $w \in \mathcal{L}(P)$ and $w \in \mathcal{L}(P')$. Since both P and P' satisfy $\mathcal{G}(\vec{m})$, we must have $(\rho \quad f_{enable}(\vec{m}) = enable(a))(P, w)$, and $(\rho \quad f_{enable}(\vec{m}) = enable(a))(P', w)$. Thus, $w :: a \in \mathcal{L}(P) \leftrightarrow w :: a \in \mathcal{L}(P')$. It follows that $\mathcal{L}(P) = \mathcal{L}(P')$. Now suppose that $\mathcal{L}(P) = \mathcal{L}(P')$, but for some $w \in \mathcal{L}(P)$, $(\rho Out)(P, w, x) \neq (\rho Out)(P', w, x)$. One of these must disagree with $(\rho f_{output})(P, w, \vec{m}, x)$. So, one of P and P′ must not satisfy $\mathcal{G}$ on $\vec{m}$, contradicting the premise. Thus, a grammar specifies at most one (black box

distinct) transducer for each vector of arguments.
EndProof.

To show that each grammar is satisfied by at least one automaton on each vector of arguments, we develop a construction of the satisfying automaton. Note that the value of $f(\vec{x})$ in (P, w) does not depend on P if $f \in \mathrm{mpr}(\mathbf{precedes})$. Similarly, if $f \in \mathrm{mpr}(\mathbf{precedes}, \mathbf{in}_0)$, then the value of $f(\vec{x})$ in (P, w) may depend only on w and on the factors of P. Write $(w\ f)(\vec{x})$ to denote the value of a $\mathrm{mpr}(\mathbf{precedes})$ function $f(\vec{x})$ under (P, w) for all P. And write $(w, \mathrm{Fact}\ f)(\vec{x})$ to denote the value of a $\mathrm{mpr}(\mathbf{precedes}, \mathbf{in}_0)$ function $f(\vec{x})$ under (P, w) for all P with factors $\mathrm{Fact} = (P_1, \ldots, P_r)$.

Definition 3.7: The congruence $\approx$.

$$w \approx u \bmod (f, \vec{m}) \Leftrightarrow (\forall z, v)(zwv\ f)(\vec{m})) = (zuv\ f)(\vec{m})$$

$$w \approx u \bmod (\mathrm{Fact}\ f, \vec{m}) \Leftrightarrow (\forall z, v)(zwv\ \mathrm{Fact}\ f)(\vec{m})) = (zuv\ \mathrm{Fact}\ f)(\vec{m})$$

The congruence class of a sequence w, with respect to some $\mathrm{mpr}(\mathbf{precedes})$ expression $f(\vec{m})$ is denoted as follows:

$$[w]_{f(m)} = \{u : u \approx w \bmod (f, \vec{m})\}$$

Similarly, the congruence class of a sequence w, with respect to some $\mathrm{mpr}(\mathbf{precedes}, \mathbf{in}_0)$ expression $f(\vec{m}$ and factor tuple Fact is denoted as follows:

$$[w]_{f(m),\mathrm{Fact}} = \{u : u \approx w \bmod (f, \vec{m}, \mathrm{Fact})\}$$

Let $A^*/(f, \vec{m}) = \{[w]_{f(m)} : w \in A^*\}$ and let $A^*/(f, \vec{m}, \mathrm{Fact}) = \{[w]_{f\ m),\mathrm{Fact}} : w \in A^*\}$.

Theorem 7: For any $f \in \mathrm{mpr}(\mathbf{precedes})$, there is an integer k, called the modal bound of $f(\vec{m})$ so that $A^*/(f, \vec{m})$ contains at most k distinct elements.

For any $f \in \mathrm{mpr}(\mathbf{precedes}, \mathbf{in}_0)$, and $\mathrm{Fact} = (P_1, \ldots, P_r)$ there is an integer k called the composite modal bound of $f(\vec{m})$ and Fact, so that $A^*/(f, \vec{m}, \mathrm{Fact})$ contains at most k distinct elements.

Proof. For $f \in PR$ the number k is 1, the trace and factors are not involved in the evaluation of $f(\vec{m})$ at all. For $\mathbf{precedes}(a, i, b, j)$ the number is at most i+j by theorem 1.

- If $f(c, x) \stackrel{\mathrm{def}}{=} (\mathbf{in}\ c\ g)(\vec{x})$ then the number of configurations of P_c is the bound.

- If $f(\vec{x}) \stackrel{\text{def}}{=} h(g_1(\vec{x}), ..., g_n(\vec{x}))$, the modal bound of each $g_i(\vec{m})$ is k_i, and the modal bound of $h(g_1(\vec{m}, ..., g_n(\vec{m}))$ is k_0 then $k \leq \prod_{i=0}^{n}(k_i + 1)$.

- If $f(0, \vec{x}) \stackrel{\text{def}}{=} g(\vec{x}), f(l+1, \vec{x}) \stackrel{\text{def}}{=} h(l, \vec{x}, f(l, \vec{x}))$, and and if k is the bound for $g(\vec{m})$ then k is the bound for $f(0, \vec{m})$. And if k_1 is the bound for $f(l, \vec{m})$, and k_2 is the bound for $f' \stackrel{\text{def}}{=} h(l, \vec{m}, f(l, \vec{m}))$, then $(k_1 + 1) * (k_2 + 1)$ is the bound. **End proof.**

EndProof

Let $A = f_{alphabet}(\vec{m})$. Let $Spec(\vec{x})$ be the conjunction of the right hand sides of the clauses of $\mathcal{G}(\vec{x})$. Then $S = A^*/(Spec, \vec{m})$ is a finite set, by the theorem we have just proved. Define $\delta : S \times A \rightarrow S$ and $\lambda : S \rightarrow O$ so that:

$$\delta([w]_{Spec\ m}, a) = \begin{cases} [w :: a]_{Spec\ m} & \textbf{if } (w\ f_{enable})(a) = 1; \\ \text{Undefined} & \textbf{otherwise.} \end{cases}$$

$$\lambda([w]_{Spec\ m}) \stackrel{\text{def}}{=} (w\ f_{output})(\vec{m}).$$

Call the resulting flat product automaton $\mathcal{M}(\mathcal{G}(\vec{m})$. The construction for a composite grammar is nearly identical. This construction proves that there is at least one transducer that satisfies $\mathcal{G}(m)$.

Lemma 2: If $\mathcal{G}(\vec{x})$ is a modal grammar, then $\mathcal{M}(\mathcal{G}(\vec{m})$ satisfies $\mathcal{G}$ on $\vec{m}$. Furthermore, $\mathcal{G}(\mathcal{M})$ is aperiodic if $\mathcal{G}$ is unitary.

To see that $\mathcal{M}(\mathcal{G}(\vec{m})$ is aperiodic note that for some i we must have $w :: a^{(i)} \approx_{Spec\ m)} w :: a^{(i+1)}$.

Finally, we want to show that there is a grammar $\mathcal{G}$ so that $\mathcal{M}(\mathcal{G}(\vec{m})$ is black-box equivalent to P for each P. Suppose that P is a product form automaton. Let P′ be a trivial product form automaton with $\mathcal{L}(P) = \mathcal{L}(P')$ and $\lambda_P(\Delta_P(w)) = \lambda_{P'}(\Delta_{P'}(w))$. Then P and P′ are equivalent. and we we can easily define a composite grammar $\mathcal{G}$ so that P satisfies $\mathcal{G}$. Let $P' = \{A, S, initial, O, \delta, \lambda\}$. Define $\mathcal{G}_0$ from $\mathcal{G}'$ as follows:

$$\mathcal{G}' \stackrel{\text{def}}{=} \{$$
$$\text{Alphabet} = S$$
$$\bigwedge \text{Out} = \begin{cases} \text{initial} & \textbf{if Initial;} \\ s & \textbf{else if}(\forall s')precedes(s', 1, s, 1). \end{cases}$$
$$\bigwedge \text{enable}(s) = 1$$
$$\}$$

Define $\mathcal{G}$ as follows:

$$
\begin{aligned}
\mathcal{G} \stackrel{\text{def}}{=} \{ & \\
& \mathsf{Alphabet} = A \\
& \bigwedge \mathsf{Components} = \{c\} \\
& \bigwedge (\mathsf{in}\, c) \Box \mathcal{G}' \\
& \bigwedge \mathsf{Out} = \lambda((\mathsf{in}\, c)\mathsf{Out}) \\
& \bigwedge \mathsf{enable}(a) = \delta((\mathsf{in}\, c)\mathsf{Out}, a) \neq \perp \\
& \}
\end{aligned}
$$

Clearly $\mathcal{G}$ specifies P'.

The two lemmas and the final construction constitute a proof of the exact grammar theorem stated below.

Theorem 8: Exact grammar theorem. If $\mathcal{G}(\vec{x})$ is an exact grammar, then for every tuple of constant arguments $\vec{m}$, there is a transducer P which satisfies $\mathcal{G}(\vec{m})$, and all transducers which satisfy $\mathcal{G}(\vec{m})$ are black-box equivalent. Furthermore, if P is a transducer, there is an exact grammar $\mathcal{G}$ and constants $\vec{m}$ so that P satisfies $\mathcal{G}(\vec{m})$.

3.4 Summary

While chapter 2 provided a formal interpretation for m.p.r. functions, the expressive power of the arithmetic was left unresolved. The expressiveness of a formal language can be measured in three ways. First, we can, informally, measure the difficulty of constructing complex specifications. In this chapter we have argued that modal function definitions, and modal grammars provide a very flexible and intuitive style of specification. To illustrate the use of these techniques we developed a specification of a storage cell, and then used this specification to construct specifications of a clocked memory bank, and a clocked fifo. The second measurement of expressive power involves the difficulty of expressing relationships between specifications. A formal method should be able to clarify our understanding of how one specification can *implement* a second specification, or how one specification can expose an abstract property that is implicit in a more detailed specification. This chapter introduces a variety of techniques for formally proving the existence of such relationships. We argue that the intuitive clarity of the semantic base, and the "open" quality of the arithmetic which allows free use of methods from the rest of mathematics, facilitate formal deduction in the arithmetic. The chapter concludes with an examination of a technical measurement of expressive power: what structures can be specified in the

arithmetic, and how precise are the specifications. We prove that every transducer can be specified by m.p.r. *exact grammar*, that exact grammars specify only transducers that are equivalent in a strong sense, and that there is an algorithm for constructing the specified transducer from an exact grammar.

CHAPTER 4

TEMPORAL LOGIC STYLE FUNCTIONALS

The temporal logics [50, 33] are modal extensions of the classical propositional and predicate logics which are intended to facilitate reasoning about dynamic systems. In the branching temporal logics [5, 16] we are concerned about the future values of propositions. For example, we write $\Diamond E$ to assert that it is inevitable that sometime in the future E will become true. The interval temporal logics [42] are more concerned about how propositions change value within finite blocks of time. For example, we write $E; F$ to assert that the current interval consists of two overlapping sub-intervals, E is true during the first sub-interval, and during the second sub-interval F stays true. Analogs to the operators of both these logics can be defined as m.p.r. functionals. For example, for every function $f(x)$, we have a boolean function $\Box f(x)$ which is true iff $f(x) > 0$ at the termination of all enabled paths. Because the semantic base of m.p.r. arithmetic is so different from that of the temporal logic, there are some technical differences between the original operators and their m.p.r analogs. For example, the usual temporal logic interpretation of $\Box\Diamond E$ is " E will be true infinitely often." Since m.p.r. arithmetic has little to say about infinite computations, we must make do with an interpretation having to do with finite cycles in a state graph. A further difference arises from the contrast between the notion of *next state* in temporal logic and the analogous notion of *after one state transition* in m.p.r. arithmetic. Despite these differences, the intuitive meaning of the temporal operators is preserved in the m.p.r. analogs.

Outline. The chapter begins by defining analogs of some of the more useful interval operators. The second section defines m.p.r. analogs of the simpler branching time operators and provides an intuitive interpretation of the temporal operators within the context of finite transducers. The third section develops proof techniques for temporal functionals, and shows that a proof system developed for temporal logic can be used within m.p.r arithmetic. The final section shows how the pumping property of finite transducers and the syntactic properties of m.p.r. functions can be used to actually

construct the temporal style operators. The material of the final section can be safely passed over by anyone interested more in the applications of m.p.r arithmetic than in the theory underlying the arithmetic.

4.1 The interval functions

A sequence of transitions u describes a computation of the system, a sequence of operations that drive the system from its current state. Each prefix of the sequence drives the system to a new configuration, so we can associate a sequence of transitions with a sequence of configurations. We are often interested in asking how many times a property holds during a sequence: how many states visited by the sequence satisfy the property. We might also want to know the sum of the values an expression takes during computation of the sequence. In this section we develop the techniques for formalizing such questions. These techniques will be employed in the following chapter, to define functionals that are analogous to the operators of the branching time logics [5].

We let $(\mathsf{sometime}\, u)\mathsf{E}$ be true iff E attains a non-zero value at some point during the computation of u. Note that E will become true during u iff some prefix $v \prec u$ leads to a state where E is true. Thus, we define $(\mathsf{sometime}\, u)\mathsf{E}$ to be $(\exists v \prec u)(\mathsf{after}\, v)\mathsf{E}$. Other interval functionals can be defined similarly. We let $(\mathsf{always}\, u)\mathsf{E}$ be true iff E is non-zero in the current state, and stays that way during the computation of u. Finally, we let $(\mathsf{cumu}\, u)\mathsf{E}$ be the *cumulative* total of E during u.

Definition 4.1: The interval operators sometime, always, and cumu .

$$(\mathsf{sometime}\, y)f(\vec{x}) \stackrel{\mathrm{def}}{=} (\exists v \prec y)(\mathsf{after}\, v)f(\vec{x}) > 0$$
$$(\mathsf{always}\, y)f(\vec{x}) \stackrel{\mathrm{def}}{=} (\forall v \prec y)(\mathsf{after}\, v)f(\vec{x}) > 0$$
$$(\mathsf{cumu}\, y)f(\vec{x}) \stackrel{\mathrm{def}}{=} \sum_{v \prec y}(\mathsf{after}\, v)f(\vec{x})$$

If we know $(\mathsf{sometime}\, u)\mathsf{E}$ to be true, we will often be interested in finding the least prefix $v \prec u$ so that $(\mathsf{after}\, v)\mathsf{E}$.

Definition 4.2: The function Upto.

$$(\mathsf{upto}\ \langle\rangle)\mathsf{E} \stackrel{\mathrm{def}}{=} \langle\rangle, (\mathsf{upto}\ aw)\mathsf{E} \stackrel{\mathrm{def}}{=} \begin{cases} \langle\rangle & \text{if } \mathsf{E} > 0 \\ \langle a\rangle \cdot (\mathsf{after}\, a)(\mathsf{upto}\ w)\mathsf{E} & \textbf{otherwise.} \end{cases}$$

Let's prove that $(\mathsf{sometime}\, u)\mathsf{E}$ implies that $(\mathsf{upto}\ u)\mathsf{E}$ is, in fact, the shortest prefix of u which reaches a state where E is true. For proof, suppose $(\mathsf{sometime}\, u)\mathsf{E}$. It follows, by definition of sometime, that there must be some shortest prefix $v \prec u$ so

that (after v)E. Suppose that $u = \langle\rangle$. In this case there is only one prefix of u, so (after$\langle\rangle$E), and thus E is true in the current state. Thus, (upto u)E $= \langle\rangle$ and the result follows. Now consider the inductive case. Suppose that $u = av$, length(u) $= k + 1$ and the result holds for all $j \leqslant k$. If E is true, then (upto u)E $= \langle\rangle$ which is certainly the least prefix satisfying E, and if E is false, then (upto u)E $= \langle a \rangle \cdot$ (upto$v$)E, and the induction hypothesis lets us conclude that the result holds. More formally:

M.p.r. theorem 10:

$$(\text{sometime}\, u)E \rightarrow \left(\begin{array}{l} (\text{after}(\text{upto}\, u)E)E \\ \bigwedge(\forall v \prec u)((\text{after}\, v)E \rightarrow (\text{upto}\, u)E \preceq v) \end{array} \right).$$

Systems are often designed so that some property cannot hold more than once during a particular sequence of transitions. In terms of intervals, we can say that (cumu u)E $\leqslant 1$. There is a useful inductive formulation of this assertion that asserts that if $u = a : w$, either $\neg$E is true in the current state, or (after a)(cumu wE) $= 0$.

M.p.r. theorem 11:

$$(\text{cumu}\, aw)E \leqslant 1 \rightarrow ((\neg E) \bigvee (\text{after}\, a)(\text{cumu}\, w)E = 0$$

Proof:

$$\begin{aligned}
&(\text{cumu}\, aw)E \\
&\quad = \{\textit{By definition}\}\textstyle\sum_{v \prec aw}(\text{after}\, v)E \\
&\quad = \{\textit{By arithmetic}\}(\text{after}\langle\rangle)E + \textstyle\sum_{av \prec aw}(\text{after}\, av)E \\
&\quad = \{\textit{By rule 2}\}E + \textstyle\sum_{av \prec aw}(\text{after}\, av)E \\
&\quad \leqslant \{\textit{Premise}\}1 \\
&\quad \Rightarrow \{\textit{By arithmetic}\}E = 0 \bigvee (\textstyle\sum_{av \prec aw}(\text{after}\, av)E = 0) \\
&\quad = \{\textit{def. of} \neg\}\neg E \bigvee (\textstyle\sum_{av \prec aw}(\text{after}\, av)E = 0) \\
&\quad = \{\textit{Rule 2}\}\neg E \bigvee (\textstyle\sum_{av \prec aw}(\text{after}\, a)(\text{after}\, v)E = 0) \\
&\quad = \{\textit{Rule 5}\}\neg E \bigvee (\text{after}\, a)(\textstyle\sum_{a:v \prec a:w}(\text{after}\, v)E = 0) \\
&\quad = \{\textit{By algebra}\}\neg E \bigvee (\text{after}\, a)(\textstyle\sum_{v \prec w}(\text{after}\, v)E) = 0) \\
&\quad = \{\textit{def.}\ \text{cumu}\}\neg E \bigvee (\text{after}\, a)((\text{cumu}\, u)E = 0)
\end{aligned}$$

EndProof: .

Some of the other interval operators that can be found in [42] can also be defined within m.p.r. arithmetic. For example, we can define ";" so that $(E;_u F) \stackrel{\text{def}}{=} (\exists v, v')(u = vv' \bigwedge(\text{always}\, v)E \bigwedge(\text{after}\, v)(\text{always}\, v')F)$. We have found **always**, **sometime**, and **cumu** sufficient for our purposes, however, and will not further investigate interval operators in this work.

4.2 Branching time

4.2.1 The operators of branching time

There are several variations on the theme of branching time [16, 54, 5], but we will concentrate on the Universal Branching (UB) logic of [5] adding only an until operator not found in UB. The notation used here is not the standard notation, however, because the standard branching time notation is appalling. The branching time logic is based on a view of a computation as a tree of possible execution sequences. Thus, the branching time operators are divided into universal operators, which assert that a property will hold in all branches, and existential operators, which assert that a property must hold in at least one branch. In standard notation **X** and **G** denote "next state" and "future states along an infinite computation path", respectively. Thus, $\forall \mathbf{X} E$ is true iff E must be true in every state reachable in one step or time unit, and $\exists \mathbf{X} E$ is true iff at least one of the "next states" satisfies E. Similarly $\forall \mathbf{G} E$ is true iff E is true in all future states, but $\exists \mathbf{G} E$ is true iff there is some infinite branch from the current state, so that E is true in every state on that branch. In preference to this notation, and to free the intuitive meaning from dependence on the model of infinite computation trees, we offer an alternative notation and intuitive interpretation.

- $\bigcirc E$ is true iff E is true in every state reachable in on step ($\forall \mathbf{X} E$).
- $\circledcirc E$ is true iff E is true in at least one state reachable in one step ($\exists \mathbf{X} E$).
- $\Box E$ is true iff E is true in every reachable future state and the present state ($\forall \mathbf{G} E$).
- $\boxdot E$ is true iff E there is a computation path which can keep E true indefinitely ($\exists \mathbf{G} E$).
- $\Diamond E$ is true iff E is inevitable, if there is a k so each path of length k or more must visit a state where E is true ($\neg \exists \mathbf{G} \neg E$).
- $\diamondsuit E$ is true iff E there is a computation path which leads to a state where E is true ($\neg \forall \mathbf{G} \neg E$).

We will also want to consider the common temporal logic binary operator until. Intuitively, E until F is true iff E must remain true along all computation paths, until a state is reached where F is true. A conceptually simple recursive description of until makes this a little more precise: E until $F \Leftrightarrow (F \bigvee (E \bigwedge \bigcirc (E \text{ until } F)))$.

4.2.2 The m.p.r. branching time functionals

The operators $\bigcirc$ and $\circledcirc$ are easily modeled by m.p.r functionals.

Definition 4.3: Next state functionals.

$$\bigcirc f(\vec{x}) \stackrel{\text{def}}{=} (\forall a \in \textbf{Alphabet}())(\textbf{enable}(a) \rightarrow (\textbf{after}\, a) f(\vec{x}))$$
$$\circledcirc f(\vec{x}) \stackrel{\text{def}}{=} (\exists a \in \textbf{Alphabet}())(\textbf{enable}(a) \bigwedge (\textbf{after}\, a) f(\vec{x}))$$

The remaining modifiers will first be formalized directly, using unbounded quantification. Since unbounded quantification is not permitted in the definition of m.p.r. functions, the reader might be justified in questioning the appropriateness of these definitions. It is, however, permissable to use unbounded quantification if we can show that it is merely a convenience, and not absolutely required for the construction. Fortunately, we are able to remove the unbounded quantification by taking advantage of the syntactic properties of m.p.r. functions, and the pumping lemma of regular languages. The skeptical reader is, thus, invited to examine section 4.4 where the pumping lemma based construction is carried out.

The modifiers $\Box$ and $\diamondsuit$ are simple.

Definition 4.4: Henceforth, and Possibly.

$$\Box f(\vec{x}) \Leftrightarrow (\forall u)(\textbf{enable}(u) \rightarrow (\textbf{after}\, u) f(\vec{x}))$$
$$\diamondsuit f(\vec{x}) \Leftrightarrow (\exists u)(\textbf{enable}(u) \bigwedge (\textbf{after}\, u) f(\vec{x}))$$

For $\diamondsuit$ and $\boxdot$, we need to make a convention about *dead-end paths*. Intuitively, $\diamondsuit E$ should mean that there is a bound k on the number of state transitions that can be traversed without visiting a state where $E > 0$. But, this does not handle the case of dead end paths, u s.t. $\textbf{enable}(u)$ and $\textbf{enable}(ua) \Leftrightarrow a = 0$. If $\diamondsuit E$ is true, we want E to be inevitable. Thus, we need to make sure that shorter dead-end paths also visit a state where E is true. For this reason we define $\text{Epaths}(i)$ to be the set of all enabled paths of length i, ***plus*** all enabled dead-end paths of length less than i.

Definition 4.5: The function Epaths.

$$\text{Epaths}(0) \stackrel{\text{def}}{=} \{\langle\rangle\},$$
$$\text{Epaths}(i+1) \stackrel{\text{def}}{=} \{ua \text{ s.t. } u \in \text{Epaths}(i) \bigwedge a \in \textbf{Alphabet}() \bigwedge \textbf{enable}(ua)\}$$
$$\{u \text{ s.t. } u \in \text{Epaths}(i) \bigwedge (\forall a \in \textbf{Alphabet}())\neg\, \textbf{enable}(ua)\}$$

Definition 4.6: Eventually and indefinitely.

$$\Diamond f(\vec{x}) \Leftrightarrow (\exists i)(\forall u \in \mathrm{Epaths}(i))(\mathbf{sometime}\, u) f(\vec{x})$$
$$\Box f(\vec{x}) \Leftrightarrow (\forall i)(\exists u \in \mathrm{Epaths}(i))(\mathbf{always}\, u) f(\vec{x})$$

Finally, we formalize E **until** F as an assertion that every enabled path u must satisfy either (**always** u)E or (**sometime** u)F. On first glance this may appear to be too weak. But because **enable**(u) implies that $(\forall v \prec u)\,\mathbf{enable}(v)$, this assertion is quite strong enough. Suppose that every path u satisfies either (**always** u)E or (**sometime** u)F. If u satisfies (**always** u)E, then our intuition about the meaning of **until** is satisfied. Suppose that $\neg$(**always** u)E. Then we must have (**sometime** u)F. Is it possible for E to become non-zero during u, *before* F becomes true? If so, then there would be some prefix $v \prec u$ so that (**sometime** v)$\neg$E and (**always** v)$\neg$F. But because because u is enabled, v must be enabled, and thus by the hypothesis, either (**always** v)E or (**sometime** v)F.

Definition 4.7: The functional until .

$$\mathrm{E}\ \mathbf{until}\ \mathrm{F} \Leftrightarrow (\forall u)(\mathbf{enable}(u) \rightarrow ((\mathbf{always}\, u)\mathrm{E} \bigvee (\mathbf{sometime}\, u)\mathrm{F}))$$

4.3 Proofs with branching functionals

If we are to be able to use the branching functionals, we require methods for analyzing expressions that contain them. Ben-Ari [5] has, conveniently, developed a set of proof rules and axioms for UB which we show to correspond to m.p.r. theorems. This correspondence makes a collection of useful theorems available, and also verifies the faithfulness of the analogy between the temporal operators and branching functionals. Thus, the first sub-section of this section examines the proof system of UB and incorporates it into m.p.r. arithmetic. The second section verifies the analogy between the temporal logic **until** and the m.p.r version, and develops some proof rules involving **until**.

4.3.1 Verification of the analogy

This section verifies the UB proof system within m.p.r. arithmetic.

Theorem 9: The rewritten proof rules of UB are proof rules for m.p.r., and the rewritten axioms are also m.p.r. proof rules.

The proof of this theorem consists of verifying the validity of the axioms or UB. The proof rules of UB are already m.p.r. proof rules. These rules are:

R1. If E is a substitution instance of a tautology then E is valid.

R2. If E is true, and if $A \rightarrow B$ is also true, then B is true. (Modus ponens).

R3. If E is valid, then $\Box E$ is valid (Necessitation).

The axioms of UB are proof rules, all of which can be proved from the m.p.r. theorems we have previously developed.

1. $\Box(E \rightarrow F) \rightarrow (\Box E \rightarrow \Box F)$
2. $\bigcirc(E \rightarrow F) \rightarrow (\bigcirc E \rightarrow \bigcirc F)$
3. $\Box E \rightarrow (\bigcirc E \wedge \bigcirc \Box E)$
4. $\Box(E \rightarrow \bigcirc E) \rightarrow (E \rightarrow \Box E)$
5. $\Box(E \rightarrow F) \rightarrow (\boxdot E \rightarrow \boxdot F)$
6. $\boxdot E \rightarrow E \wedge \circledcirc \boxdot E$

Some re-written UB theorems.

1. $\Box E \rightarrow \boxdot E$
2. $\Box(E \rightarrow \circledcirc E) \rightarrow (E \rightarrow \boxdot E)$
3. $\Box E \rightarrow A$
4. $\Box E \rightarrow \Diamond E$
5. $\bigcirc(E \rightarrow F) \rightarrow (\circledcirc E \rightarrow \circledcirc F)$
6. $\Box(E \rightarrow F) \rightarrow (\Diamond E \rightarrow \Diamond F)$
7. $\boxdot E \rightarrow \circledcirc E$
8. $\bigcirc E \rightarrow \circledcirc E$
9. $\Box(E \wedge F) \leftrightarrow \Box E \wedge \Box F$
10. $\Diamond(E \wedge F) \rightarrow \Diamond E \wedge \Diamond F$
11. $\bigcirc(E \wedge F) \rightarrow (\bigcirc E \wedge \bigcirc F)$
12. $\Box E \wedge \boxdot F \rightarrow \boxdot (E \wedge F)$

13. $\Box E \leftrightarrow E \wedge \bigcirc \Box E$
14. $\boxdot E \leftrightarrow E \wedge \circledcirc \boxdot E$
15. $\Box E \leftrightarrow \Box \Box E$
16. $\boxdot E \leftrightarrow \boxdot \boxdot E$
17. $\boxdot (E \rightarrow \bigcirc E) \rightarrow (E \rightarrow \boxdot E)$
18. $\Diamond \Box E \rightarrow \Box \Diamond E$
19. $\boxdot ((A \vee \boxdot F) \wedge (\boxdot E \vee F) \leftrightarrow (\boxdot E \vee \boxdot F)$
20. $\bigcirc \Box E \leftrightarrow \Box \bigcirc E$
21. $\circledcirc \boxdot E \leftrightarrow \boxdot \circledcirc E$

4.3.2 Proof of the UB axioms

1. $\Box(E \rightarrow F) \rightarrow (\Box E \rightarrow \Box F)$

$$
\begin{aligned}
\Box(E \rightarrow F) &= \{ \textit{ by definition of } \Box \} \\
&\quad (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u)(E \rightarrow F)) \\
&= \{ \textit{ by independence } \} \\
&\quad (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u)E \rightarrow (\mathbf{after}\, u F))) \\
&= \{ \textit{ by boolean algebra } \} \\
&\quad (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u)E \wedge \mathbf{enable}(u) \rightarrow (\mathbf{after}\, u F))) \\
&\Rightarrow \{ \textit{ by quantifier algebra } \} \\
&\quad (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u)E) \rightarrow (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u F)) \\
&\Rightarrow \{ \textit{ by definition of } \Box \} \\
&\quad \Box E \rightarrow \Box F
\end{aligned}
$$

2. $\bigcirc(E \rightarrow F) \rightarrow (\bigcirc E \rightarrow \bigcirc F)$

$$
\begin{aligned}
\bigcirc(E \rightarrow F) &= \{ \textit{ def. of } \bigcirc \} \\
&\quad (\forall a)(\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a)(E \rightarrow F)) \\
&= \{ \textit{ state independence } \} \\
&\quad (\forall a)(\mathbf{enable}(a) \rightarrow ((\mathbf{after}\, a E) \rightarrow (\mathbf{after}\, a F))) \\
&= \{ \textit{ boolean algebra } \} \\
&\quad (\forall a)((\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a E)) \rightarrow (\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a F))) \\
&\Rightarrow \{ \textit{ boolean algebra } \} \\
&\quad (\forall a)(\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a E)) \rightarrow (\forall a)(\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a F)) \\
&\Rightarrow \{ \textit{ def. } \bigcirc \} \bigcirc E \rightarrow \bigcirc F
\end{aligned}
$$

3. $\Box E \rightarrow (\bigcirc E \wedge \bigcirc \Box E)$

$\Box E$ = { *by definition of* $\Box$ } $(\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))$
⇒ { *by algebra* }
$(\forall au)(\mathbf{enable}(au) \rightarrow (\mathbf{after}\, au E))$
= { *path division* }
$(\forall au)(\mathbf{enable}(au) \rightarrow (\mathbf{after}\, a)(\mathbf{after}\, u E))$
= { *def.* $\mathbf{enable}^*$ }
$(\forall au)(\mathbf{enable}(a) \wedge (\mathbf{after}\, a\, \mathbf{enable})(u) \rightarrow (\mathbf{after}\, a)(\mathbf{after}\, u E))$
⇒ { *boolean algebra* }
$(\forall au)(\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))))$
= { *distribution* }
$(\forall au)(\mathbf{enable}(a) \rightarrow (\mathbf{after}\, a(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))))$
⇒ { *algebra* }
$(\forall a)(\mathbf{enable}(a) \rightarrow (\forall u)(\mathbf{after}\, a(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))))$
= { *def.* $\bigcirc$ } $\bigcirc((\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E)))$
= { *def.* $\forall$, *boolean algebra* }
$\bigcirc((\mathbf{enable}(\langle\rangle) \rightarrow (\mathbf{after}\langle\rangle)E) \wedge (\forall u)\, \mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E)))$
= { *def.* $\mathbf{enable}^*$, *path division* } $\bigcirc(E \wedge (\forall u)\, \mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E)))$
= { *def.* $\Box$ }
$\bigcirc(E \wedge \Box E)$

4. $\Box(E \rightarrow \bigcirc E) \rightarrow (E \rightarrow \Box E)$

{*Induction base case* }
$(\mathbf{after}\langle\rangle(E \rightarrow \bigcirc E)$
= {*path division* }E
⇒ { *boolean algebra* }E
⇒ { *path division* }$(\mathbf{after}\langle\rangle)E$

{*Induction step* }
$(\forall u, \mathrm{length}(u) < k)(\mathbf{enable}(u) \wedge (\mathbf{after}\, u)(E \rightarrow \bigcirc E)$
$\rightarrow (E \rightarrow (\forall u, \mathrm{length}(u) < k)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))$

{*Induction step premise* }
$\mathbf{enable}(ua)$
⇒ { *def. of* $\mathbf{enable}^*$ } $\mathbf{enable}(u)$
{*Induction hypothesis* }
⇒ $(E \rightarrow (\forall u, \mathrm{length}(u) < k)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))$
⇒ {*Premise* }$(\mathbf{after}\, u)(E \rightarrow \bigcirc E)$
⇒ {*by algebra* }$(\mathbf{after}\, u \bigcirc E))$
⇒ {*definition of* $\bigcirc$ }$(\mathbf{after}\, ua E))$

⇒ {*from the induction* }$E \rightarrow (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{after}\, u E))$
= {*definition of* $\Box$ }$E \rightarrow \Box E$

5. $\Box(E \rightarrow F) \rightarrow (\boxdot E \rightarrow \boxdot F)$

$$
\begin{aligned}
&\Box(E \rightarrow F) \\
&\quad = \{\textit{Definition}\}(\forall u)(\mathsf{enable}(u) \rightarrow (\mathsf{after}\,u)(E \rightarrow F) \\
&\quad = \{\textit{independence}\} \\
&\quad (\forall u)(\mathsf{enable}(u) \rightarrow (\mathsf{after}\,u)E \rightarrow (\mathsf{always}uF)) \\
&\quad = \{\textit{algebra, def. of}\ \mathsf{Epaths}\} \\
&\quad (\forall i)(\forall u \in \mathsf{Epaths}(i))((\mathsf{after}\,u)E \rightarrow (\mathsf{always}uF)) \\
&\quad = \{\textit{algebra}\} \\
&\quad (\forall i)(\exists u \in \mathsf{Epaths}(i))((\mathsf{after}\,u)E \rightarrow (\mathsf{always}uF)) \\
&\quad = \{\textit{algebra}\} \\
&\quad (\forall i)(\exists u \in \mathsf{Epaths}(i))((\mathsf{after}\,u)E \\
&\quad \rightarrow (\forall i)(\exists u \in \mathsf{Epaths}(i))((\mathsf{after}\,u)F \\
&\quad = \{\textit{Definition}\}\ \boxdot E \rightarrow \boxdot F
\end{aligned}
$$

6. $\boxdot E \rightarrow E \wedge \bigcirc\boxdot E$
(essentially the same proof as above)

4.3.3 Until

In this section we prove that our version of until satisfies the usual recursive specification, and also derive some general proof rules involving until.

Proof that until satisfies its recursive definition.

Suppose that E until F is true. Then, by definition, we have $(\forall u)(\mathsf{enable}(u) \rightarrow (\mathsf{always}\,u)E \vee (\mathsf{sometime}\,u)F)$. We can rewrite this to distinguish the empty path from the other paths as follows:

$$
\begin{aligned}
&\mathsf{enable}(\langle\rangle) \rightarrow (\mathsf{always}\langle\rangle)E \vee (\mathsf{sometime}\langle\rangle)F \\
&\wedge(\forall au)(\mathsf{enable}(au) \rightarrow (\mathsf{always}\,au)E \vee (\mathsf{sometime}\,au)F).
\end{aligned}
$$

The first clause can be greatly simplified because $\mathsf{enable}(\langle\rangle) = 1$, $(\mathsf{always}\langle\rangle)E = E$, and $(\mathsf{sometime}\langle\rangle)F = F$. We have shown, therefore:

$$
\begin{aligned}
&(E \vee F) \\
&\wedge(\forall au)(\mathsf{enable}(au) \rightarrow (\mathsf{always}\,au)E \vee (\mathsf{sometime}\,au)F).
\end{aligned}
$$

Now we simplify the second clause, by splitting the empty prefix case from both the always and sometime.

$$
\begin{aligned}
(E \vee F) \wedge(\forall au)(\mathsf{enable}(au) \rightarrow\ & (E \wedge (\mathsf{after}\,a)(\mathsf{always}\,u)E) \\
& \vee (F \vee (\mathsf{after}\,a)(\mathsf{sometime}\,u)F).
\end{aligned}
$$

If F is false, then E must be true, so we conclude:

$$
\begin{aligned}
F \vee (E \wedge (\forall au)(\mathsf{enable}(au) \rightarrow\ & ((\mathsf{after}\,a)(\mathsf{always}\,u)E) \\
& \vee (\mathsf{after}\,a)(\mathsf{sometime}\,u)F).
\end{aligned}
$$

Since $\mathbf{enable}(au) = (\mathbf{enable}(a) \bigwedge (\mathbf{after}\ a)\mathbf{enable})(u)$ we get:

$$F \bigvee (E \bigwedge (\forall a)(\mathbf{enable}(a) \rightarrow (\mathbf{after}\ a)\left((\forall u)\,\mathbf{enable}(u) \rightarrow \begin{array}{l}(\mathbf{after}\ a)(\mathbf{always}\ u)E \\ \bigvee (\mathbf{after}\ a)(\mathbf{sometime}\ u)F.\end{array}\right)$$

This reduces, by definition, to $F \bigvee (E \bigwedge \bigcirc(E\ \mathbf{until}\ F))$ which is the recursive definition.

Proofs involving until.

An important m.p.r. theorem allows us to prove $E\ \mathbf{until}\ F$ *inductively* on paths. That is, we show that if in all future states, E implies that $F \bigvee \bigcirc(E \bigvee F)$ is true, then we must have $E \rightarrow (E\ \mathbf{until}\ F)$.

M.p.r. theorem 12: Until induction.

$$\Box((E \rightarrow (F \bigvee \bigcirc(E \bigvee F))) \rightarrow E\ \mathbf{until}\ F$$

To prove m.p.r. theorem 12 we must show that $E\ \mathbf{until}\ F$ follows from E and the premise. By definition of until this reduces to a proof that $E \rightarrow (\forall u)(\mathbf{enable}(u) \rightarrow (\mathbf{always}\ u)E \bigvee (\mathbf{sometime}\ u)F)$. We can prove this by induction. Suppose that $u = \langle\rangle$, then $E \rightarrow (\mathbf{always}\langle\rangle)E$, so the inductive case is true. Now suppose the theorem is true for all w so that $\mathrm{length}(w) \leqslant k$. Consider some wa so that $\mathbf{enable}(wa)$. We know that $E \rightarrow ((\mathbf{always}\ w)E \bigvee (\mathbf{sometime}\ w)F)$ from the induction hypothesis. If $(\mathbf{sometime}\ w)F$ we have nothing to prove. Suppose $\neg(\mathbf{sometime}\ w)F$. Then we must have $(\mathbf{always}\ w)E$. But we also know that $(\mathbf{after}\ w)(E \rightarrow (F \bigvee \bigcirc(E \bigvee F))$. Since, $(\mathbf{always}\ w)E$ we conclude that $(\mathbf{after}\ w)(F \bigvee \bigcirc(E \bigvee F))$. Thus, either $(\mathbf{always}\ wa)E$ or $(\mathbf{after}\ wa)F$ which implies $(\mathbf{sometime}\ wa)F$.

EndProof.

Synchronization of devices is often accomplished by letting one device wait until the second device is ready. The next m.p.r. theorem states that if an expression E is non-zero, and $E\ \mathbf{until}\ (E \bigwedge F)$, and $(\mathbf{sometime}\ u)F$, then if u is enabled, $(\mathbf{sometime}\ u)(E \bigwedge F)$. That is, if device X signals its intention to synchronize until both its own signal and the signal of device Y are sensed, and if the signal from Y will be sensed during the same interval, we can conclude that the synchronization will take place.

M.p.r. theorem 13: Until latching.

$$\Box((E\ \mathbf{until}\ (E \bigwedge F) \bigwedge \mathbf{enable}(u) \bigwedge (\mathbf{sometime}\ u)F \rightarrow (\mathbf{sometime}\ u)(E \bigwedge F)).$$

The proof of m.p.r. theorem 13 is straightforward. We assume the theorem hypothesis. From $(\text{sometime}\,u)F$ there must be some $v_0 \prec u$, so that $(\text{after}\,v_0)F$. We note that, by definition of until, every enabled path u must satisfy $(\text{always}\,u)E \bigvee (\text{sometime}\,u)(E \bigwedge F)$. If the second case is true, we are done. If $(\text{always}\,u)E$ is true, then for every $v \prec u$, we must have $(\text{after}\,v)E$. Thus, $(\text{after}\,v_0)F \bigwedge (\text{after}\,v_0)E$. Writing $\bigwedge$ in functional form, we have $\text{and}((\text{after}\,v_0)E, (\text{after}\,v_0)F)$. Since and is state independent we can move it down into the scope of the after and conclude $(\text{after}\,v_0)\text{and}(E, F)$, i.e., $(\text{after}\,v_0)(E \bigwedge F)$. **EndProof.**

4.4 Pumping theorems

In a finite state system, we expect that assertions about "all enabled paths" or, "an unbounded enabled path", can be reduced to assertions about all paths of a certain length, or a path of a certain length, simply because of the pumping lemma. That is, if a property is true for all enabled paths, then we note that all enabled paths of over a certain length must traverse cycles in the finite state graph, and thus are *pumped instances* of shorter paths. We can, therefore, reduce the question to a question about a finite number of enabled paths. Because the function precedes depends on the trace, not the system configuration, pumping is a little more complex for *contexts* than it is for standard finite transducers. There is a pumping property, however, and we will develop it in this section, and use it to properly define the temporal functionals. There is a great deal of overlap between the proofs here, and those in the section on exact grammars. There is enough difference, however, that these proofs are self-contained.

Notation. A sequence α is a subsequence of u iff $u = v\alpha v'$ for some v and v'. A subsequence is *non-trivial* iff it is not the empty sequence, and not the sequence $\langle 0^{(i)} \rangle$ (the null transition is not interesting).

Definition 4.8: The function Nontriv.

$$\text{Nontriv}(\langle\rangle) \stackrel{\text{def}}{=} 0, \text{Nontriv}(aw) \stackrel{\text{def}}{=} \begin{cases} 1 & \textbf{if } a \neq 0; \\ \text{Nontriv}(w) & \textbf{otherwise.} \end{cases}$$

4.4.1 A congruence based on configurations

Definition 4.9: The configuration congruence $\sim$.

$$u \sim v \bmod P \Leftrightarrow (\forall w, z \in A^*)(\Delta(wuz) = \Delta(wvz))$$

Theorem 10: If length(u) > (ρ Pump_number)(P, w), for arbitrary w, then there is some α so that u = vαz and Nontriv(α), and u ~ vα^(i)z mod P.

Proof: This is simply a reformulation of the classical pumping lemma. There are only (ρ Pump_number)(P, w) states, thus, u must traverse repeated states. Note, that Pump_number( ) is the product state set cardinality, not just the number of configurations. Just because Δ(w) = Δ(u) does not imply that u ~ w mod P. But, if w and u cause P to enter the same state, then they will be congruent.

4.4.2 A congruence based on relative transition ordering

There is a second congruence relation that we can define on paths. Note that (ρ precedes)(P, w, a, i, b, j) depends solely on the i rightmost a's and j rightmost b's in w. We call i the a *depth* of precedes(a, i, b, j), and similarly, call j the b depth. We call functions d : A → Nat, *depth bounds* and use these functions to define a second congruence. The sequences u and v will be equivalent under w and d : A → Nat iff RelOrder(u, a, i, b, j) = RelOrder(v, a, i, b, j) for whenever i ⩽ d(a) and j ⩽ d(b).

Definition 4.10: The depth congruence ≈.

$$
\begin{aligned}
u \approx v \bmod\ & (A, w, d) \\
& \Leftrightarrow (\forall a, b \in A)(\forall i \leqslant d(a))(\forall j \leqslant d(b))(\forall z \in A^*) \\
& \mathrm{RelOrder}(wuz, a, i, b, j) = \mathrm{RelOrder}(wvz, a, i, b, j)
\end{aligned}
$$

Note that if (∀a)d(a) ⩽ d'(a), u ≈ v mod (A, w, d') ⇒ u ≈ v mod (A, w, d).

Theorem 11: If u contains at least $k > \sum_a d(a)$ non-trivial subsequences $\alpha_1, \ldots, \alpha_k$ so that $u = u_1, \alpha_1, \ldots, u_k, \alpha_k$, then for at least one of those subsequences, α_r: $u = v\alpha_r v'$

$$(\forall i)(u \approx v\alpha_r^{(i)}v' \bmod (A, w, d)$$

Proof. For each a, mark each α_i as a- *interesting* iff α_i contains at least one a, and there are fewer than d(a) a's in $u_i, \ldots, u_k, \alpha_k$. Since we can mark at most d(a) α's for each a, we can mark at most $\sum_a d(a)$ α's in total. Thus, there must be at least one α which is completely unmarked. Pick one unmarked subsequence β so that u = vβv'. Now consider the difference between place(wuz, a, i) and place(wvβ^(k)v'z) for i < d(a). If there are at least i a's in v'z then:

$$\mathrm{place}(wv\beta^{(k)}v'z) = \mathrm{place}(wuz, a, i) + \mathrm{length}(v\beta^{(k)}).$$

If there are fewer than i a's in v'z, then β must contain no a's. Otherwise, β would be marked as a- *interesting*. Thus place(wvβ^(k)v'z, a, i) = place(wuz, a, i).

Suppose that $RelOrder(wv\beta^{(i)}v') = 1$. Then $place(wuz, a, i) < place(wuz, b, j)$. Note that if there are at least i a's in $v'z$, there must be at least j b's in $v'z$ for this inequality to hold. If $place(wuz, b, j) = place(wv\beta^{(k)}v'z, b, j)$, then $place(wuz, a, i) = place(wv\beta^{(k)}v'z, a, a)$ and the inequality holds for the pumped string. If $place(wuz, b, i) = place(wv\beta^{(k)}v'z, b, j) + length(\beta^{(k)}v'z)$ then the inequality must also hold for the pumped string. **EndProof.**

We now want to find a method for deriving a depth bound function for an arbitrary m.p.r. function. We define a functional $D : MPR \rightarrow MPR$ so that $d_{(P,w,f,\vec{m})}(a) = (\rho(Df))(P, w, a, \vec{m})$ is a depth bound function.

Definition 4.11: The depth bound functional D.

If $f(x) \stackrel{def}{=} 0$

$(Df)(x,a) \stackrel{def}{=} 1$

If $f(x) \stackrel{def}{=} x+1$

$(Df)(x,a) \stackrel{def}{=} 1$

If $f(y,\vec{x}) \stackrel{def}{=} \vec{x}_y$

$(Df)(a,y,\vec{x}) \stackrel{def}{=} 1$

If $f() \stackrel{def}{=} \mathbf{Alphabet}()$

$(Df)(a) \stackrel{def}{=} 1$

If $f() \stackrel{def}{=} \mathbf{oUtputs}()$

$(Df)(a) \stackrel{def}{=} 1$

If $f() \stackrel{def}{=} \mathbf{Components}()$

$(Df)(a) \stackrel{def}{=} 1$

If $f() \stackrel{def}{=} \mathbf{Pump\_number}()$

$(Df)(a,x) \stackrel{def}{=} 1$

If $f() \stackrel{def}{=} \mathbf{Out}_i()$

$(Df)(a,x) \stackrel{def}{=} 1$

If $f(x) \stackrel{def}{=} \mathbf{enable}(x)$

$(Df)(a,x) \stackrel{def}{=} 1$

If $f(x) \stackrel{def}{=} \mathbf{precedes}(x,y,x',y')$

$$(Df)(a,x,y,x',y') \stackrel{def}{=} \begin{cases} \max\{y,y'\} & \textbf{if } a = x \wedge a = x'; \\ y & \textbf{elseif } a = x \wedge a \neq x'; \\ y' & \textbf{elseif } a = x' \wedge a \neq x; \\ 1 & \textbf{otherwise.} \end{cases}$$

If $f(x) \stackrel{def}{=} \mathbf{f\_effect}(x,y)$

$(Df)(a,x,y) \stackrel{def}{=} 1$

If $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}),\ldots,g_n(\vec{x}))$

$(Df)(a,\vec{x}) \stackrel{def}{=} (Dh)(a,\ldots g_i(\vec{x}),\ldots) * \prod_i (Dg_i)(a,\vec{x})$

If $f(0,\vec{x}) \stackrel{def}{=} g(\vec{x}), f(r+1,\vec{x}) \stackrel{def}{=} h(r,\vec{x}), f(r,\vec{x}))$

$(Df)(a,0,\vec{x}) \stackrel{def}{=} (Dg)(a,\vec{x})$

$(Df)(a,r+1,\vec{x})) \stackrel{def}{=} (Dh)(a,r,\vec{x},f(a,r,\vec{x})) * (Df)(a,r,\vec{x})$

If $f(y,\vec{x}) \stackrel{def}{=} (\mathbf{after}\, y)g(\vec{x})$

$(Df)(a,y,\vec{x}) \stackrel{def}{=} (\mathbf{after}\, y)(Dg)(a,\vec{x})$

If $f(y,\vec{x}) \stackrel{def}{=} (\mathbf{in}\, y)g(\vec{x})$

$(Df)(a,y,\vec{x}) \stackrel{def}{=} 1$

Note that D is non-decreasing in the complexity of the the function argument. If $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}),..,g_n(\vec{x}))$, then $(Df)(a,\vec{x}) \geqslant (Dh)(a,g_1(\vec{x})...,g_n(\vec{x}))$ and $(Df)(a,\vec{x}) \geqslant (Dg_i)(a,\vec{x})$. Similarly, if $f(\vec{x})$ is defined by primitive recursion from h and g, we must have $(Df)(a,0,\vec{x}) \geqslant (Dg)(a,\vec{x})$ and $(Df)(a,r+1,\vec{x}) \geqslant (Dh)(a,r,\vec{x},f(r,\vec{x}))$.

Thus, if $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}),....,g_n(\vec{x}))$ then

$$u \approx v \bmod (A,w,d_{[P,w,x]}) \rightarrow \begin{array}{l} u \approx v \bmod (A,w,d_{[P,w,h,...g_i(x)...]}) \\ \bigwedge u \approx v \bmod (A,w,d_{[P,w,g_i,x]}) \end{array}$$

4.4.3 The intersection of two congruences

Theorem 12:

$$\begin{array}{l} u \approx v \bmod (A,w,d_{[P,w,f,x]}) \bigwedge u \sim v \bmod P \\ \rightarrow (\rho f)(P,wuz,\vec{x}) \stackrel{def}{=} (\rho f)(P,wvz,\vec{x}) \end{array}$$

Proof. Since the p.r. functions are independent of the context, the theorem is trivially true for the initial functions that are p.r. initial functions. For enable and f_effect, it is sufficient to note that we must have $u \sim v \bmod P$. For precedes we note that $u \approx v \bmod (P,w,\text{precedes},\vec{m})$, and so:

$$\begin{array}{l} (\rho\,\text{precedes})\ (P,wuz,\vec{m}) = \text{RelOrder}(wuz,\vec{m}) \\ \qquad = \text{RelOrder}(wvz,\vec{m}) \\ \qquad = (\rho\,\text{precedes})(P,wvz,\vec{m}). \end{array}$$

Suppose that the theorem is true for h, g, and g_i.

- If $f(\vec{x}) \stackrel{def}{=} h(g_1(\vec{x}),...,g_n(\vec{x}))$ then:

$$\begin{array}{l} (\rho f)\ (P,wuz,\vec{x}) = (\rho h)(P,wuz,...(\rho g_i)(P,wuz,\vec{x})...) \\ \qquad = (\rho h)(P,wuz,...(\rho g_i)(P,wvz,\vec{x})...) \\ \qquad = (\rho h)(P,wvz,...(\rho g_i)(P,wvz,\vec{x})...) \\ \qquad = (\rho f)(P,wvz,\vec{m}) \end{array}$$

- If $f(0,\vec{x}) \stackrel{def}{=} g(\vec{x}), f(r+1,\vec{x}) \stackrel{def}{=} h(r,\vec{x},f(r,\vec{x}))$ then:

$$\begin{array}{l} (\rho f)(P,wuz,0,\vec{x}) = (\rho g)(P,wuz,\vec{x}) = (\rho f)(P,wvz,0,\vec{x}) \\ (\rho f)\ (P,wuz,r+1,\vec{x}) = (\rho h)(P,wuz,(\rho f)(P,wuz,r,\vec{x})) \\ \qquad = (\rho h)(P,wuz,(\rho f)(P,wvz,r,\vec{x})) \\ \qquad = (\rho h)(P,wvz,(\rho f)(P,wvz,r,\vec{x})) \\ \qquad = (\rho f)(P,wvz,r+1,\vec{m}) \end{array}$$

- If $f(y,\vec{x}) \stackrel{def}{=} (\text{after}\,y)f(\vec{x})$, then:

$$\begin{array}{l} (\rho f)\ (P,wuz,y,\vec{x}) \\ \qquad = (\rho g)(P,wuzy,\vec{x}) \\ \qquad = (\rho g)(P,wvzy,\vec{x}) \\ \qquad = (\rho f)(P,wvz,y,\vec{x}) \end{array}$$

- If $f(y,\vec{x}) \stackrel{\text{def}}{=} (\text{in}\, y) f(\vec{x})$, then:

$$\begin{aligned}(\rho f)\ (P, wuz, y, \vec{x}) &= (\rho g)(P_y, wuz, \vec{x}) \\ &= (\rho g)(P_y, wvz, \vec{x}) \\ &= (\rho f)(P, wvz, \vec{x})\end{aligned}$$

EndProof.

Definition 4.12: The depth bound functional plen.

$$(\text{plen}f)(\vec{x}) \stackrel{\text{def}}{=} \sum_a ((Df)(a,\vec{x}+1) * (\textbf{Pump\_number}() + 1).$$

Theorem 13:

$$\begin{aligned}&\text{length}(u) \geqslant (\text{plen}f)\vec{x}) \\ &\rightarrow (\exists v, w, \alpha)\left(\begin{array}{l} v\alpha w = u \bigwedge \text{Nontriv}(\alpha) \\ \bigwedge(\textbf{enable}(v\alpha^{(i)}v) = \textbf{enable}(u) \\ \bigwedge(\text{after}\, v\alpha^{(i)}w) f(\vec{x}) w) = (\text{after}\, u) f(\vec{X})\end{array}\right)\end{aligned}$$

Proof: The sequence u must contain at $k > (Df)(\vec{x})$ nontrivial sequences $\alpha_1, ..., \alpha_k$ so that each sequence is pumpable for the $\sim$ equivalence — the subsequence begins and ends at the same configuration. But, $k > \sum_a (Df)(a,\vec{x})$ nontrivial sequences must contain at least one subsequence that can be pumped while preserving $\approx$. **EndProof.**

Corollary. 4:

$$\begin{aligned}&(\forall z, \text{length}(z) \leqslant (\text{plen}f)(\vec{x}))\ \textbf{enable}(z) \rightarrow (\text{after}\, z) f(\vec{x})) \\ &\rightarrow \\ &(\forall u)(\textbf{enable}(u) \rightarrow (\text{after}\, u) f(\vec{x}))\end{aligned}$$

Proof: Suppose the premise is true. Then if $\text{length}(u) \leqslant (\text{plen}f)(\vec{x})$ the conclusion must follow trivially. If not, then u must contain a pumpable non-trivial subsequence which can be unpumped, to produce a shorter sequence. Continue this process until we obtain a sequence u′ of length bounded by $(\text{plen}f)(\vec{x})$. Then, by the premise, $(\text{after}\, u') f(\vec{x})$, but, $(\text{after}\, u') f(\vec{x}) = (\text{after}\, u) f(\vec{x})$. **EndProof.**

Corollary. 5:

$$\begin{aligned}&(\exists z, \text{length}(z) = (\text{plen}f)(\vec{x}))\ \textbf{enable}(z) \bigwedge (\textbf{always}\, z) f(\vec{x})) \\ &\rightarrow \\ &(\exists z, \text{length}(z) = i))\ \textbf{enable}(z) \bigwedge (\textbf{always}\, z) f(\vec{x}))\end{aligned}$$

Proof: Suppose the premise is true. Then if $i \leqslant (\mathsf{plenf})(\vec{x})$ the conclusion must follow trivially. If not, then consider some u satisfying the premise. Since u must contain a pumpable non-trivial subsequence which can be pumped, we can easily produce a sequence $v\alpha^{(i)}z$ such that $u = v\alpha z$, $(\mathbf{after}\, v\alpha^{(i)}z)f(\vec{x})$ and $\mathsf{length}(v\alpha^i z) = i + k$ for $k \leqslant \mathsf{length}(\alpha)$. Clearly, $(\mathbf{always}\, v\alpha^{(i)}z)f(\vec{x})$. This completes the proof. **EndProof.**

Definition 4.13: The path functionals.

$$\Box f(\vec{x}) \stackrel{\mathbf{def}}{=} (\forall u \in \mathsf{Epaths}((\mathsf{plenf})(\vec{x})))(\mathbf{after}\, u)f(\vec{x})$$
$$\boxdot f(\vec{x}) \stackrel{\mathbf{def}}{=} (\exists u \in \mathsf{Epaths}((\mathsf{plenf})(\vec{x})))(\mathbf{always}\, u)f(\vec{x})$$
$$\diamondsuit f(\vec{x}) \stackrel{\mathbf{def}}{=} (\exists u \in \mathsf{Epaths}((\mathsf{plenf})(\vec{x})))(\mathbf{sometime}\, u)f(\vec{x})$$
$$\Diamond f(\vec{x}) \stackrel{\mathbf{def}}{=} (\forall u \in \mathsf{Epaths}((\mathsf{plenf})(\vec{x})))(\mathbf{sometime}\, u)f(\vec{x})$$
$$(f\ \mathbf{until}\ g)(\vec{x}) \stackrel{\mathbf{def}}{=} (\forall u \in \mathsf{Epaths}((\mathsf{plen}(g \bigwedge f))(\vec{x}))) \left(\begin{array}{l} (\mathbf{always}\, u)g(\vec{x}) \\ \bigvee (\mathbf{sometime}\, u)f(\vec{x}) \end{array} \right)$$

Theorem 14: The pumping lemma definition of the temporal functionals agree with the unbounded definitions given in definitions 4.4, 4.6, and 4.7

Proof

- The "only if" part of the implication ($\rightarrow$) is trivial for $\Box$. Suppose $\Box f(\vec{x})$and **enable**(u).If $\mathsf{length}(u) \leqslant (\mathsf{plenf})(\vec{x})$ then the implication is obvious. Suppose $\mathsf{length}(u) > (\mathsf{plenf})(\vec{x})$. Then, u must be $f(\vec{x})$ equivalent to some sequence v of length bounded by i. It follows that $v \in \mathsf{Epaths}(i)$ or v is a prefix of some element of $\mathsf{Epaths}(i)$. Thus, $(\mathbf{always}\, v)f(\vec{x})$, and $(\mathbf{always}\, u)f(\vec{x})$.

- For $\Diamond$, the if implication ($\leftarrow$) is trivial. Suppose $\Diamond f(\vec{x})$, and consider the value i which bounds the lengths of paths. If $i \leqslant (\mathsf{plenf})(\vec{x})$, then obviously $\Diamond f(\vec{x})$. If $i > (\mathsf{plenf})(\vec{x})$, each path in $\mathsf{Epaths}(i)$ must of the form $u \cdot v$ so that $(\mathbf{after}\, u)f(\vec{x})$. Either $\mathsf{length}(u) \leqslant i$ or u is $f(\vec{x})$ equivalent to a a shorter path.

- For $\boxdot$, the only if implication ($\rightarrow$) is trivial. Suppose $\boxdot f(\vec{x})$, and note this gives us at least one enabled path u, with $\mathsf{length}(u) = (\mathsf{plenf})(\vec{x})$ and $(\mathbf{always}\, u)f(\vec{x})$. Clearly u must contain a non-trivial subsequence α which can be pumped to produce enabled paths v of length $i + (j * \mathsf{length}(\alpha))$ so that $(\mathbf{always}\, v)f(\vec{x})$.

- For $\diamondsuit$, the if implication ($\leftarrow$) is trivial. Suppose $\diamondsuit f(\vec{x})$, and consider the u so that $(\mathbf{after}\, u)f(\vec{x})$. Clearly can be unpumped if it is too long.

- Now, lets prove the second unbounded **until** is equivalent to the bounded definition. Suppose E **until** F. By the hypothesis we must have $\mathbf{enable}(u) \rightarrow (\mathbf{always}\, u)E \bigvee (\mathbf{sometime}\, u)F$ for u such that $\mathrm{length}(u) \leqslant (\mathrm{plen}(f \bigwedge g))(\vec{x})$. Thus every enabled sequence is congruent to a sequence which satisfies the conditions.

4.5 Summary

Temporal logic has captured the interest of many researcher in the field of formal methods because it has an intuitively appealing language for describing some of the most important properties of systems. This chapter has shown that the temporal modal modifiers can be simulated by m.p.r. functionals. We have shown that the simulation preserves the intuitive content of the originals, but provides a sharper and more tractable semantic interpretation. In particular, the interval operators, which require extensions to standard temporal logic formulations, are simulated using the same small set of m.p.r. primitives that we use to simulate the branching time operators. And, we show that there is a constructive, bounded semantics for eventuality and "forever" which can replace the infinitary or unbounded semantics of the temporal logics.

CHAPTER 5

REAL TIME

Put simply, a system is *real-time* if and only if the passage of physical time causes state changes. This definition differs from more standard definitions of real-time (c.f. [59])in that it makes no reference to deadlines or other *externally observed* measures of time. It would be possible to extend the m.p.r. model so that contexts included a clock; one could, for example, speak of the value of an expression in the context of (P, w, time). The extension would allow us to consider the duration of a state transition. For example, a state transition which represents a positive signal on a wire might require some time to pass to account for "rise time." But adding a clock to the context would require abandoning the pure state machine semantics of the arithmetic. That is, the state of a system would no longer be determined solely by a finite state machine and its trace. As a result, both specifications and the mathematical foundations of the arithmetic would become a great deal more complex. Fortunately, the notion of an external clock is not necessary for detailed investigation of real-time. Instead, we can consider certain state transitions to represent the passage of time, and then count the number of instances of these transitions in the trace. For example, a real-time queue transducer might traverse a push.p transition in response to a command, and then traverse some number of tick transitions before returning to an idle state. More simply, a wire transducer might output value 1 if at least t tick transitions have been traversed since the most recent raise transition. The tick transitions are intended to represent the passage of physical time, not the signals of some imaginary global clock device. If the state machine represents a real-time sensitive system, then the passage of physical time must cause it to change state. Thus, our definition of real-time is motivated by our desire to remain within a pure finite state transition semantics.

In this section we discuss methods for investigating real-time systems, and look at two examples. This chapter begins by developing some techniques for describing real-time systems. We then consider contrasting examples. The first

example is a real-time priority queue. The queue example is intended to illustrate close analysis of real-time constraints and the process of refining a real-time specification to get a more detailed specification of an implementation. The second example is a small fragment of the Futurebus+ arbitration protocol. This example, will illustrate specification of a distributed, asynchronous, real-time algorithm.

5.1 Real time techniques

In this chapter, we will always define systems with alphabets that contain transition symbol tick to represent the passage of one unit of time. The amount of time that passes during computation of a sequence is, therefore, equivalent to the number of tick symbols in the path.

Definition 5.1: The function Tcount.

$$\mathsf{TCount}(u) \stackrel{\text{def}}{=} \begin{cases} 0 & \textbf{if } u = \langle\rangle; \\ \mathsf{TCount}(u') & \textbf{if } (\exists u')(\exists a \neq \mathsf{tick})u = u' \cdot \langle a \rangle; \\ \mathsf{TCount}(u') + 1 & \textbf{if } (\exists u')u = u' \cdot \langle \mathsf{tick} \rangle. \end{cases}$$

Definition 5.2: The function duration.

$$\mathsf{duration}(u) \stackrel{\text{def}}{=} \mathsf{TCount}(u) * \mathbf{enable}(u).$$

Thus, duration(u) > 0 implies that u is an enabled path, during which duration(u) time units pass.

Each state transition that is not a tick transition represents an instantaneous control state change. A system with tick $\in$ Alphabet() is defined to satisfy *real-time liveness* if and only if there is a bound on the number of instantaneous control transitions that can be traversed without a tick transition intervening. That is, a system satisfies real-time liveness iff there is some k so that enable(u) and Length(u) > k implies that TCount(u) > 0. Intuitively, a real-time system which is not real-time live makes little sense. Such a system could traverse an unbounded number of state changes instantaneously. If a system is real-time live, we we can define m.p.r. functionals which check all paths u where duration(u) $\leqslant$ n. In particular, we can define specialized real-time versions of $\Box$ and $\Diamond$ which depend on the number of ticks which are traversed by a path. Intuitively, $\Box_t f(x)$ should be true if and only if f(x) > 0 for at least the next t time units. Similarly, $\Diamond_t f(x)$ should be true if and only if f(x) > 0 *within* at

most t time units. These functionals are analogous to the real-time temporal modifiers found in [30].

Definition 5.3: Timed versions of henceforth and eventually.

$$\Box_t f(x) \stackrel{\text{def}}{=} (\forall u)(\mathbf{enable}(u) \bigwedge \mathbf{duration}(u) \leqslant t \rightarrow (\mathbf{after}\, u) f(x))$$
$$\Diamond_t f(x) \stackrel{\text{def}}{=} (\forall u)(\mathbf{duration}(u) \geqslant t \rightarrow (\mathbf{sometime}\, u) f(x))$$

We can prove some elementary facts about $\Diamond_t$ and $\Box_t$ that will be useful in gaining intuition about their behavior.

M.p.r. theorem 14: Elementary results about $\Diamond_n$ and $\Box_n$.

1. $(\Diamond_n E \bigwedge \Box(E \rightarrow F)) \rightarrow \Diamond_n F$
2. $(\Box_{n-1} E \bigwedge \mathbf{enable}(\mathrm{tick}) \rightarrow (\mathbf{after}\, \mathrm{tick} \Box_n E)$
3. $\Box \Diamond_n E \bigwedge \mathbf{duration}(u) \geqslant k * n \rightarrow (\mathbf{cumu}) E \geqslant k$

Proof. For 14.1 we simply note that $\mathbf{duration}(u) \geqslant n$ implies that for some $v \prec u$ we have $(\mathbf{after}\, v)E$. By the premise, we have $(\mathbf{after}\, v)(E \rightarrow F)$, and thus, we have shown $(\mathbf{after}\, v)F$. For the second part of the theorem we note that $\mathbf{duration}(u) \leqslant n+1 \rightarrow (\mathbf{always}\, u)E$. Suppose that $(\mathbf{after}\, \mathrm{tick})\mathbf{duration}(v) \geqslant n$. Then $\mathbf{enable}(\mathrm{tick})$ implies that $\mathbf{duration}(\mathrm{tick} \cdot v) \geqslant n+1$. Thus $(\mathbf{always}\, \mathrm{tick} \cdot v)E$, and $(\mathbf{after}\, \mathrm{tick})(\mathbf{always}\, u)E)$. To prove m.p.r. theorem 14.3 divide u into $(k+1)$ segments, each of which contains at least n ticks.

Less trivially, consider a system where property E becomes true and stays true until both E and F are true. For example, a "handshake" protocol requires one device to signal its request and maintain the request until a second device signals its response. Formally, we write such a requirement as $\Box \Diamond_m (E \,\mathbf{until}\, (E \bigwedge F))$. That is, every m time units E will become true and stay true until $(E \bigwedge F)$ becomes true. Now suppose that F becomes true every n time units: $\Box \Diamond_n F$. It should follow that within $n + m$ time units both E and F will be true.

M.p.r. theorem 15: Duration upper bound.

$$\Box(\Diamond_m (E \,\mathbf{until}\, (E \bigwedge F)) \bigwedge \Diamond_n F) \rightarrow \Diamond_{n+m} (E \bigwedge F))$$

Proof:

1. $\Diamond_m(E \text{ until } (E \wedge F))$
2. $\Box\Diamond_n F$
3. { *Supposition* }$\text{duration}(u) > n + m$
4. { *abbreviation introduction* }$v = (\text{upto} u E)$
5. $\Rightarrow$ { *from 1,3* }$(\text{after } v)(E \text{ until } (E \wedge F))$
6. $\Rightarrow$ { *from 5,1* }$\text{duration}(v) \leqslant m$
7. { *abbreviation introduction for* z }$u = vz$
8. $\Rightarrow$ { *from 5* }$(\text{after } v)\text{duration}(z) \geqslant n$
9. $\Rightarrow$ { *from 8,2* }$(\text{after } v)(\text{sometime } z)F$
10. $\Rightarrow$ {*Rule 13, 9,5* }$(\text{after } v)(\text{sometime } z)(E \wedge F)$
11. $\Rightarrow (\text{sometime } u)(E \wedge F)$.

For a related theorem consider the case where condition E interferes with handling of condition F. For example, if E is a request which has higher priority than F. In this case, we may want to specify that E must periodically stay false long enough for $(F \wedge \neg E)$ to become true. We write $\Diamond_t\Box_n\neg E$ to assert that within t time units the system will enter an interval of duration n, during which E will stay false. Suppose that $\Box\Diamond_m F$ is true, and that $\Diamond_t\Box_n\neg E$ is also true, for $n > m$. Thus, within t time units, E will be false for at least m time units. Within those m time units, F must become true. So these two conditions ensure that $\Diamond_{t-m}(\neg E \wedge F)$.

M.p.r. theorem 16: Duration lower bound.

$$(n > m \wedge \Diamond_t\Box_n\neg E \wedge \Box\Diamond_m F) \rightarrow \Diamond_{t-m}(\neg E \wedge F))$$

The proof here is a simple consequence of the definitions. Suppose that $\text{duration}(u) \geqslant m + t$. There is some prefix $v \prec u$ so that $(\text{after } v)\Box_n\neg F)$ and $\text{duration}(v) \leqslant t$. This is a consequence of $\Diamond_t\Box_n\neg E$.9 But, $(\text{after } v)\Diamond_m F)$ must be true because of the premise $\Box\Diamond_m F$. Since v contains at most t ticks, the remainder of u, that is z s.t. $vz = u$, must contain at least $n + t - t = n$ ticks. Thus, z will contain at least m ticks. Since u is enabled, $(\text{after } v)\,\text{enable}(z)$, and thus $(\text{after } v)\text{duration}(z) \geqslant m$. Thus, $(\text{after } v)(\text{sometime } z)F$. Because $(\text{after } v)\Box_n(\neg E)$, and $n > m$, we conclude that $(\text{after } v)(\text{sometime } z)(\neg E \wedge F)$.

5.2 A real-time priority queue

The queue device described in this section can be considered to be part of a system which uses time-dated, perishable information. Packets of information

labeled with a *laxity* are provided to the queue device, which then makes the most urgent packet — the packet with the smallest laxity — available for output. As real-time passes, the queue device should update the latencies of its packets and discard packets which expire (have 0 latencies). The packets may be job dispatch requests, datagrams , or any other information that must either be delivered before a real-time deadline, or must be disposed of. The queue has three tasks:

- Accept packets from the environment,
- Make the "most urgent" packet available for dequeueing,
- Periodically update the queue, discarding expired packets.

Packets are pairs $p = (d, l)$ where $d = data(p)$ is the packet data, and $l = lat(p)$ is the integer *laxity*, defining the number of time units that can pass before the packet expires. We suppose that $data(p)$ and $lat(p)$ have finite ranges, so that the set P of all possible packets, is also finite.

5.2.1 The specification

We begin by listing the possible inputs accepted by the queue device. These consist of commands to enqueue or dequeue a packet, and an input that indicates the passage of one unit of real-time.

$$\text{Alphabet} = \{enq.p, deq, tick : p \in P\}$$

The queue device has 4 output functions.

$$\text{Outputs} = \{Ready, Full, Empty, First\}$$

The device will output a boolean value to inform the environment of whether or not it is *ready* for a new command. Thus, we give the device an output function Ready, and let the parameter resp_time constrain its behavior.

$$Ready : \emptyset \rightarrow \{0, 1\}$$
$$\Diamond_{resp\_time} Ready$$

Ready will act in conjunction with Empty and Full to control when command inputs are enabled.

$$enable(a) = \left(\begin{array}{l} (a = enq.p \bigwedge Ready \bigwedge \neg Full) \\ \bigvee (a = deq \bigwedge Ready \bigwedge \neg Empty) \\ \bigvee (a = tick) \end{array} \right)$$

We now specify the "logical" operation of the queue in terms of the behavior of a function Q. The value of Q at any moment will be the current sequence of queued packets. The sequence will always be sorted in order of laxity, so that the rightmost packets have the least non-zero laxity. In order to describe insertion of packets, in order, we define a function q_enq as follows.

$$q\_enq(q,p) \stackrel{def}{=} \begin{cases} \langle p \rangle & \text{if } q = \langle\rangle; \\ \langle p \rangle \cdot q & \text{if } q = \langle p' \rangle \cdot q' \bigwedge lat(p) \geqslant lat(p'); \\ \langle p' \rangle \cdot q\_enq(q', p) & \text{if } q = \langle p' \rangle \cdot q' \bigwedge lat(p) < lat(p'). \end{cases}$$

Packets are "dequeued" by removing the rightmost packet, thus the value of the queue after a dequeue operation will be rtrunc(Q). Finally, the queue is *updated*, by decrementing the laxities of all enqueued packets, and discarding those which expire.

$$q\_update(q) \stackrel{def}{=} \begin{cases} \langle\rangle & \text{if } q = \langle\rangle; \\ \langle (d, l) \rangle \cdot q\_update(q') & \text{if } q = \langle (d, l+1) \rangle \cdot q'; \\ q\_update(q') & \text{if } q = \langle (d, 1) \rangle \cdot q'. \end{cases}$$

$$\begin{array}{l} Q : \emptyset \rightarrow Sequences \\ Initial \rightarrow Q = \langle\rangle \\ \Box(\textbf{after}\ enq.p)Q = q\_enq(Q) \\ \bigwedge \Box(\textbf{after}\ deq)Q = rtrunc(Q) \\ \bigwedge \Box((\textbf{after}\ tick)Q = Q \bigvee (\textbf{after}\ tick)Q = q\_update(Q)) \\ \bigwedge \Box\Diamond_{upd\_time + upd\_drift}(\textbf{after}\ tick)Q = q\_update(Q) \\ \Box((\textbf{after}\ tick)Q = q\_update(Q) \rightarrow \Diamond_1 \Box_{upd\_time}(\textbf{after}\ tick)Q = Q)) \end{array}$$

The operation of Full, Empty, and First will be defined in terms of Q and Ready.

$$\begin{array}{l} Ready \rightarrow (Empty \leftrightarrow (Q = \langle\rangle)) \\ Ready \rightarrow (Full \leftrightarrow (length(Q) = MAX\_Q) \\ Ready \rightarrow (First = head(Q)) \end{array}$$

Note that the values of these output functions are not constrained at all when Ready = 0. We only specify what will happen if the queue device is used correctly. If the queue is used when it it not Ready, then it will enter an undefined state. the device, then the specification does not describe the result.

We collect all the constraints, and define the interface presented by the queue to the environment in a complete specification.

$$
\begin{array}{l}
\text{packet\_queue}(MAX\_Q, upd\_period, upd\_drift, resp\_time) \stackrel{\text{def}}{=} \\
\{\ (C1)\quad \textbf{Alphabet} = \{enq.p, deq, tick : p \in P\} \\
\quad (C2)\quad \textbf{Outputs} = \{Ready, Full, Empty \in \{0,1\}; First \in P\} \\
\\
\quad (C3)\quad \textbf{enable}(a) = \left(\begin{array}{l} (a = enq.p \bigwedge Ready \bigwedge \neg Full) \\ \bigvee (a = deq \bigwedge Ready \bigwedge \neg Empty) \\ \bigvee (a = tick) \end{array} \right) \\
\\
\quad Q : \emptyset \rightarrow Sequences \\
\quad (C4)\quad Initial \rightarrow Q = \langle\rangle \\
\quad (C5)\quad (\textbf{after}\ a)Q \stackrel{\text{def}}{=} \begin{cases} q\_enq(Q,p) & \textbf{if}\ a = enq.p \\ rtrunc(Q) & \textbf{if}\ a = deq \end{cases} \\
\quad (C6)((\textbf{after}\ tick)Q = Q) \bigvee ((\textbf{after}\ tick)Q = q\_update(Q)) \\
\quad (C7)\quad \Diamond_{upd\_time - upd\_drift}((\textbf{after}\ tick)Q = q\_update(Q)) \\
\quad (C8)\quad ((\textbf{after}\ tick)Q = q\_update(Q)) \rightarrow \Box_{upd\_time}((\textbf{after}\ tick)Q = Q) \\
\\
\quad Ready : \emptyset \rightarrow \{0,1\} \\
\quad (C9)\quad \Diamond_{resp\_time} Ready \\
\\
\quad (C10)\quad Ready \rightarrow (Empty \leftrightarrow (Q = \langle\rangle)) \\
\quad (C11)\quad Ready \rightarrow (Full \leftrightarrow (length(Q) = MAX\_Q)) \\
\quad (c12)\quad Ready \rightarrow (First = head(Q)) \\
\}
\end{array}
$$

Figure: 5.1 A complete high level specification of the real-time queue

5.2.2 Implementation

The specification given in the previous sections defines precisely how we want the real-time priority queue to behave, but does not give any clues as to how we might implement such a queue. When we want to implement a complex system, we often use the technique of "factoring" the system into more tractable parts, and then implementing the parts independently. Much of the complexity of the high level queue specification comes from the timing of updates. In this section we will show how to factor the previous specification into two connected components: a timer, and a queue module that updates only when it receives

an interrupt from the timer. Thus, the task of deciding *when* to update enqueued packets will be given to the clock component, and the queue_module will be directed to update the queue at appropriate times. We might want to further iterate this process, by implementing the queue_module as a linked list of memory cells, but we will not do so here.

5.2.3 The timer

A device specified by timer(n, d) will count ticks up to n with d representing a possible *drift* or inaccuracy in counting. The alphabet of a timer is {tick, reset}. The timer output is a single boolean function TimedOut. The value of TimedOut is true only if at least n ticks have passed since the most recent reset, and must become true if at least n + d ticks have passed since the most recent reset. We assert this by first asserting that a reset makes TimedOut false, and then asserting that after a reset no enabled path of duration less than n will cause TimedOut to become true. We then assert that every enabled path of duration greater than n + d must make TimedOut true at least once. Finally, we ensure that once TimedOut is true, it remains true until reset.

$$
\begin{aligned}
&\mathrm{timer}(n, d) \stackrel{\mathrm{def}}{=} \{ \\
&\mathbf{Alphabet} = \{\mathrm{tick}, \mathrm{reset}\} \\
&\mathbf{Outputs} = \{\mathrm{TimedOut}\} \\
&(\mathbf{after}\,\mathrm{reset})\Box_n \neg \mathrm{TimedOut} \\
&\Diamond_{n+d}\mathrm{TimedOut} \\
&(\mathbf{after}\,\mathrm{tick})\mathrm{TimedOut} \gtrdot \mathrm{TimedOut} \\
&\mathbf{enable}(\mathrm{tick}) = 1 \\
&\mathbf{enable}(\mathrm{reset}) = \mathrm{TimedOut} \\
&\}
\end{aligned}
$$

Figure: 5.2 The timer specification.

5.2.4 The queue module

Essentially, the queue module is the same as the packet queue defined in the first section, except that it now has an explicit command to update the queue. Thus, $(\mathbf{after}\,\mathrm{tick})Q_1 = Q_1$, the passage of time does not trigger any changes to the queue. Furthermore, the last clause of the specification ensures that if the queue module is Ready, it stays Ready until it is given a command. The

complete queue, in contrast, can start an update without receiving a command from the environment. If we compare the queue module specification to the high level queue specification, we see that the module is much simpler, because much of the timing has been "factored out" of the module.

$$
\begin{array}{l}
\mathrm{queue\_mod}(MAX\_Q, \tau) \stackrel{\text{def}}{=} \{ \\
\quad \textbf{Alphabet} = \{upd, enq.p, deq, tick : p \in P\} \\
\quad \textbf{Outputs} = \{Ready, Full, Empty \in \{0,1\}; First \in P\} \\
\quad \textbf{enable}(a) = \left(\begin{array}{l} (a = enq.p \wedge Ready \wedge \neg Full) \\ \vee (a = deq \wedge Ready \wedge \neg Empty) \\ \vee (a = upd \wedge Ready) \\ \vee (a = tick) \end{array} \right) \\
\quad Initial \rightarrow (Q_1 = \langle\rangle) \\
\quad (\textbf{after}\ a)Q_1 \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} q\_enq(Q_1, p) & \textbf{if}\ a = enq.p \\ rtrunc(Q_1) & \textbf{if}\ a = deq \\ q\_update(Q_1) & \textbf{if}\ a = upd \end{array} \right. \\
\quad Ready \wedge \neg Empty \rightarrow First = head(Q_1) \\
\quad Ready \rightarrow (Full \leftrightarrow length(Q_1) = MAX\_Q) \\
\quad Ready \rightarrow (Empty \leftrightarrow Q_1 = \langle\rangle) \\
\quad (\textbf{after}\ tick)Ready \geqslant Ready \\
\quad \Diamond_\tau Ready \\
\}
\end{array}
$$

Figure: 5.3 The queue module.

5.2.5 Putting it together

A composite specification of a packet queue is given in figure 5.4.

$$
\begin{array}{l}
\mathsf{pqsystem}(MAX\_Q, t, d, r) \stackrel{\mathbf{def}}{=} \\
\{ \\
\textbf{Alphabet} = \{enq.p, deq, tick : p \in P\} \\
\textbf{Outputs} = \{Ready, Full, Empty \in \{0,1\}; First \in P\} \\
\textbf{Components} = \{clock, queue\} \\
(\textbf{in}\ clock)\Box timer)(t, d) \\
(\textbf{in}\ queue)\Box queue\_mod)(MAX\_Q, r) \\
First = (\textbf{in}\ queue)First \\
Ready = \neg TimedOut \bigwedge (\textbf{in}\ queue)Ready \\
TimedOut = (\textbf{in}\ queue)TimedOut \\
UpdTime = (TimedOut \bigwedge (\textbf{in}\ queue)Ready) \\
\textbf{enable}(a) = \left(\begin{array}{l} (a = enq.p \bigwedge Ready \bigwedge \neg Full) \\ \bigvee (a = deq \bigwedge Ready \bigwedge \neg Empty) \\ \bigvee (a = tick) \end{array} \right) \\
Q = (\textbf{in}\ queue)Q_1 \\
\qquad \textbf{f\_effect}(a, c) = \left\{ \begin{array}{ll} \langle enq.p \rangle & \textbf{if } a = enq.p \bigwedge c = queue; \\ \langle deq \rangle & \textbf{else if } a = deq \bigwedge c = queue; \\ \langle reset, tick \rangle & \textbf{else if } a = tick \bigwedge c = clock \bigwedge UpdTime; \\ \langle tick \rangle & \textbf{else if } a = tick \bigwedge c = clock; \\ \langle upd, tick \rangle & \textbf{else if } a = tick \bigwedge c = queue \bigwedge UpdTime; \\ \langle tick \rangle & \textbf{else if } a = tick \bigwedge c = queue \bigwedge \neg UpdTime; \\ \langle\rangle & \textbf{otherwise.} \end{array} \right. \\
\}
\end{array}
$$

Figure: 5.4 Specification of the implementation.

The specification begins by defining an alphabet and output set identical to that of the "high level" specification. We continue by defining the names of the two components, clock and queue. To specify the behavior of these two components, we write (in clock)(□timer(t, d) and (in queue)□queue_mod(MAX_Q, r). Note that (in clock)□timer refers to the function □timer within the sub-system named clock. Thus, we are asserting that within the subsystem named clock the function □timer will be non-zero on arguments utime and udrift. This is only true if the function timer is non-zero on arguments utime and udrift in every reachable state of the clock sub-system. Informally, (in clock)□timer(t, d) asserts that the component named clock obeys the specification timer(t, d).

We define the operation of the output functions in terms of the output functions of the components. When we write First = (in queue)First we are asserting that operation of the top level function First identical to that of the function First of component queue. Similarly, we say the system has TimedOut iff the clock has timed out:

$$\mathsf{TimedOut} = (\mathsf{in\ clock})\mathsf{TimedOut}.$$

The top level queue is Ready iff both the queue is Ready, and the clock has not TimedOut. Finally, the system should force the queue to update iff the clock has TimedOut and the queue is Ready. The enabling rules of the system can now be described clearly.

The remaining task is to coordinate the behavior of the components and to define the input to the components. Recall that the m.p.r. function f_effect(u,i) gives the sequence of transitions that component i will follow when the composite system follows u. The function f_effect (for *feedback effect*)provides the key to synchronizing components. We want enq.p and deq to be passed directly to the queue module. Thus:

$$\mathsf{f\_effect}(\mathsf{enq.p}, \mathsf{queue}) = \langle \mathsf{enq.p} \rangle,$$

and

$$\mathsf{f\_effect}(\mathsf{deq}, \mathsf{queue}) = \langle \mathsf{deq} \rangle.$$

On the other hand, enq.p and deq should have no effect on the clock.

$$\mathsf{f\_effect}(\mathsf{deq}, \mathsf{clock}) = \mathsf{f\_effect}(\mathsf{enq.p}, \mathsf{clock}) = \langle\rangle.$$

The effect of a tick on the components will depend on whether or not the clock has timed out, and whether or not the queue is ready. If both of these conditions are met, we want the tick to cause the clock to reset and the queue to begin an update. The specification of the clock is as follows:

$$\mathsf{f\_effect}(\mathsf{tick}, \mathsf{clock}) = \begin{cases} \langle \mathsf{reset}, \mathsf{tick} \rangle & \textbf{if } \mathsf{TimedOut} \bigwedge (\mathsf{in\ queue})\mathsf{Ready} \\ \langle \mathsf{tick} \rangle & \textbf{otherwise.} \end{cases}$$

Note that a reset is always accompanied by a tick. The tick operation of the composite system indicates that one unit of real-time has passed, the reset operation of the clock is considered to take no time. In order to keep the time consistent, whenever time passes on the level of the composite system, a time unit must pass for each component. Thus, we also make sure that upd commands to the queue are always accompanied by ticks.

Our next task is to show that the composite specification, implements the high level specification. That is, we want to show that any system which satisfies the composite specification will also satisfy the high level specification.

5.2.6 Verification.

We would like to show that if a system satisfies the specification of the composite implementation, it must also satisfy the high level specification. If we consider pqsystem and packet_queue as boolean functions which are conjunctions of all their clauses, then we must show that if every clause of pqsystem is always true, then every clause of packet_queue must also be always true. Formally, our task is to show that for some n,d,r:

$$\textbf{MainProposition}\ \mathsf{Initial} \bigwedge \Box \mathsf{pqsystem}(\mathsf{MAX\_Q}, n, d, r) \rightarrow \Box \mathsf{packet\_queue}(\mathsf{MAX\_Q}, \mathsf{utime}, \mathsf{udrift}, \mathsf{resp\_time})$$

We can prove each of the clauses figure 5.1 separately. Most of the clauses can be proved quite simply. Clauses 1-3 are repeated in the definition of pqsystem so are true by the algebraic rule $E \rightarrow E$. Clauses C10 – c12 are also straightforward.

(C10) $\mathsf{Ready} \rightarrow (\mathsf{Empty} \leftrightarrow (Q = \langle\rangle))$

Proof

1 {*Premise* }Ready
2 $\Rightarrow$ {*by definition* }(in queue)Ready
3 $\Rightarrow$ { *from* queue_mod }(in queue)$(\mathsf{Empty} \leftrightarrow Q_1 = \langle\rangle)$
4 $\Rightarrow$ { *distribution* }
$(\text{in queue})\mathsf{Empty} \leftrightarrow (\text{in queue})Q_1 = \langle\rangle$
5 $\Rightarrow$ { *definition* }
$\mathsf{Empty} \leftrightarrow Q = \langle\rangle$

EndProof

(C11) $\mathsf{Ready} \rightarrow (\mathsf{Full} \leftrightarrow (\mathsf{length}(Q) = \mathsf{MAX\_Q})$
Proof{ *essentially the same as above* }

(c12) $\mathsf{Ready} \rightarrow (\mathsf{First} = \mathsf{head}(Q))$**Proof**{ *essentially the same as above* }

Clauses C4 – C6 are also straightforward. We omit C4 and C5, but now give the proof of C6 which follows an interesting lemma.

Lemma 3:

$$\mathsf{UpdTime} \rightarrow (\textbf{after}\ \mathsf{tick})Q = \mathsf{upd\_q}(Q)$$
$$\bigwedge \neg \mathsf{UpdTime} \rightarrow (\textbf{after}\ \mathsf{tick})Q = Q$$

Proof

{*Assume* }UpdTime
(after tick)Q =
{pqsystem }(after tick)(in queue)Q_1
= {*inversion* }(in queue)(after f_effect(queue, tick))Q_1
= {*premise,* pqsystem }(in queue)(after⟨upd, tick⟩)Q_1
= {*path division* }(in queue)(after upd)(after tick)Q_1
= {queue_mod }(in queue)(after upd)Q_1
= {queue_mod }(in queue)(upd_q(Q_1))
= {*Since* upd_q *is p.r.* }upd_q((in queue)Q_1)
= {pqsystem }upd_q(Q)

{*Assume* }¬UpdTime
(after tick)Q =
{pqsystem }(after tick)(in queue)Q_1
= {*inversion* }(in queue)(after f_effect(queue, tick))Q_1
= {*premise,* pqsystem }(in queue)(after tick)Q_1
= {queue_mod }(in queue)Q_1
= {pqsystem }Q

EndProof

(C6) (after tick)Q = upd_q(Q) ∨ (after tick)Q = Q
Proof
{ *Follows directly from lemma 3* }

The remaining clauses C7, C8, and C9, are the most difficult, because they depend on the interaction of the clock and the queue module. Rather than going into the details of all of these, we will just look at C7 in depth. Similar methods can be used for the other clauses.

From the specification pqsystem we can easily show that:

$$\text{enable(a)} \rightarrow \begin{pmatrix} \text{(in queue) enable(f\_effect(a, queue)} \\ \wedge \text{(in queue) enable(f\_effect(a, clock)} \end{pmatrix}$$

We also note that ticks are induced in the factors only when the composite system follows a tick, and that:

$$\text{TCount(u)} = \begin{pmatrix} \text{(in queue)TCount(f\_effect(u, queue)} \\ = \text{(in queue)TCount(f\_effect(u, clock)} \end{pmatrix}$$

Using these two observations we can conclude that:

$$\text{Duration(u)} = \begin{pmatrix} \text{(in queue)Duration(f\_effect(u, queue)} \\ = \text{(in clock)Duration(f\_effect(u, clock)} \end{pmatrix}$$

From these observations, and from examining f_effect, we can conclude that:

$$(\text{in queue}\Diamond_t \text{Ready}) \rightarrow \Diamond_t(\text{in queue})\text{Ready}$$
$$\bigwedge(\text{in clock}\Diamond_t \text{TimedOut}) \rightarrow \Diamond_t(\text{in clock})\text{TimedOut}$$
$$\bigwedge(\text{in clock}\Box_t \neg\text{TimedOut}) \rightarrow \Box_t(\text{in clock}\neg\text{TimedOut}).$$

It follows immediately that:

Lemma 4:

$$\Box\left(\begin{array}{l}\Diamond_r(\text{in queue})\text{Ready} \\ \bigwedge \Diamond_{n-d}(\text{in clock})\text{TimedOut} \\ \bigwedge \text{f\_effect}(\text{clock}, \text{tick}) = \langle\text{reset}, \text{tick}\rangle \\ \qquad\qquad \rightarrow (\textbf{after tick}\Box_{n-1}(\text{in clock}\neg\text{TimedOut})\end{array}\right).$$

Leaving this lemma for the moment, consider what happens when the clock times out. From boolean algebra we know that:

$$(\text{in clock})\text{TimedOut} \rightarrow \left(\begin{array}{l}(\text{in clock})\text{TimedOut} \bigwedge(\text{in queue})\text{Ready} \\ \bigvee(\text{in clock})\text{TimedOut} \bigwedge(\text{in queue}\neg\text{Ready})\end{array}\right)$$

If the second clause of this disjunction is true, then f_effect(a, clock) cannot contain a reset. Thus, every enabled transition will lead to a state where (in clock)TimedOut is still true.

$$(\text{in clock})\text{TimedOut} \rightarrow \left(\begin{array}{l}(\text{in clock})\text{TimedOut} \bigwedge(\text{in queue})\text{Ready} \\ \bigvee \bigcirc(\text{in clock})\text{TimedOut}\end{array}\right)$$

This assertion is of the form needed to apply m.p.r theorem 12. We use this theorem to conclude:

Lemma 5:

$$(\text{in clock})\text{TimedOut} \rightarrow$$
$$(\text{in clock})\text{TimedOut}\ \textbf{until}\ (\text{in clock})\text{TimedOut} \bigwedge(\text{in queue})\text{Ready}$$

Using the two lemmas we can now derive a key timing property:

1. { *From lemma4* }$\Diamond_{n+d}$(in clock)TimedOut
2. { *From lemma4* }$\Box\Diamond_r$(in queue)Ready
3. { *From lemma5* }
   (in clock)TimedOut →
   (in clock)TimedOut **until** (in clock)TimedOut $\bigwedge$(in queue)Ready
4. ⇒ {*by m.p.r. Th15*}
   $\Diamond_{n+d+r}$(in queue)Ready $\bigwedge$(in clock)TimedOut
5. ⇒ {*From the spec* pqsystem }$\Diamond_{n+d+r}$UpdTime
6. ⇒ {*From lemma3* }
   $\Diamond_{n+d+r}$(**after** tick)Q = update_q(Q)

Thus, if we constrain $n+d+r+1$ to be less than upd_time+upd_drift we have proved clause C7:

$$(C7) \quad \Diamond_{upd\_time+upd\_drift}(\textbf{after}\ tick)Q = q\_update(Q)$$

The other clauses can be derived similarly. We end up with a collection of constraints on n, r, and d which express the relationship between the performance of the implementation and the requirements of the high-level specification.

5.3 A fragment of Futurebus+ arbitration

The Futurebus+ arbitration protocol is designed to allow for a consensus to be reached among asynchronous real-time devices. Given n devices connected to a common bus, we need to make sure that there is never more than one device which believes itself to be the "bus-master" at any instant of time. The Futurebus+ solution to this problem is complicated by the desire of the Futurebus+ designers to avoid the use of clock signals on the bus. With a clock signal, we could easily arbitrate, say, by letting each device take ownership of the bus during a window (time domain arbitration). In this section we will specify the Futurebus arbitration algorithm, but not prove it. The proof is not difficult, and uses the techniques described in the previous section and in the next chapter. The interesting part here is deriving a formal representation of a distributed real-time algorithm.

Notation Alert. In this section w is used as a variable over names of *wires*, not as the name of a trace.

We can model a circuit that has k *pins* with a transducer that has tick transitions to mark the passage of time, and sense.(b,p) transitions for $b \in \{0,1\}$ and each pin identifier p. We assume that tick represents the smallest interval of time in which a signal change can change state. That is, fluctuations on pin logic levels which have a duration less than the time measured by a tick can be ignored. We might let the time units be measured in nano-seconds, pico-seconds, or even femto-seconds, if it seems necessary. A transition sense.(1,p) represents the *instantaneous signal rise* on pin p. Likewise, sense.(0,p) represents an instantaneous signal drop. The instantaneous signal level is given by level(p) which is described in figure 5.5 Note that the initial value of level(p) is not

constrained.

$$\mathbf{precedes}(sense.(0,p),1,sense.(1,p),1) \rightarrow Level(p) = 1$$
$$\mathbf{precedes}(sense.(1,p),1,sense.(0,p),1) \rightarrow Level(p) = 0$$

Figure: 5.5 Pin logic levels

Usually, circuits cannot respond to instantaneous logic level changes. A new pin input requires a certain amount of time before it propagates through internal logic and causes a state change. Thus, we also need to be able to meaure how long a signal has stayed *stable*

$$Stable(p,t) \stackrel{\text{def}}{=} (\forall b)\,\mathbf{precedes}(sense.(b,p),1,tick,t)$$

Figure: 5.6 Signal stability

The Futurebus+ arbitration bus connects N devices to W wires. We name the pins on the bus (w, d) where w is the wire and d is the connecting device. The bus output on each pin is given by the value of $PinOut : N \times W \rightarrow \{0,1\}$. The outputs and inputs don't have to match: The output reflects the value that the bus is asserting on the pin; the input reflects the value that the bus senses on the pin. The key property of the bus is that it acts as an *or-gate* on the input signals on each wire. If all the input pins for wire w stay low for enough time, then the bus will output 0 on each pin connected to that wire. If at least one pin is forced high for long enough, then the bus will output 1 on every pin connected to that wire. The time required for this function is called the propagation time T_{prop} of the bus.

$$ArbBus(N, W, T_{prop}) =$$
$$\{$$
$$\mathbf{Alphabet} = \{tick, sense.(b,p) : b \in \{0,1\}, p \in \{0,...W-1\} \times \{0,...,N-1\}\}$$
$$\mathbf{Outputs} = \{PinOut\}$$
$$(\forall w)\left(\begin{array}{l}(\forall d)Level((w,d)) = 0 \bigwedge Stable((w,d)T_{prop}) \\ \rightarrow (\forall d')PinOut((w,d')) = 0\end{array}\right)$$
$$(\forall w))\left(\begin{array}{l}(\exists d)Level((w,d)) = 1 \bigwedge Stable((w,d)T_{prop}) \\ \rightarrow (\forall d')PinOut((w,d')) = 1\end{array}\right)$$
$$\mathbf{enable}(tick) = 1$$
$$\mathbf{enable}(sense.(p,b)) = Level(p) \neq p \bigwedge Stable(p,1)$$
$$\}$$

Figure: 5.7 The arbitration bus

Note that the bus behavior is undefined if signal changes are not separated by at least one tick, or if the bus senses two consecutive changes of the same polarity.

We now turn to the devices which will compete over the arbitration bus. Each device connects to W wires, and has an output given by the function WireOut: $W \rightarrow \{0,1\}$. The *internal delay* of a device is the time that the device requires to respond to changes in input levels. The *priority* of the device is a number $0 \leqslant \mathrm{pri} < 2^W$ which can be expressed as a W bit signal on the bus.

$$
\begin{aligned}
&\mathsf{ArbDev}(N, W, T_{prop}, T_{int}, \mathsf{pri}) = \\
&\{ \\
&\textbf{Alphabet} = \{\mathsf{tick}, \mathsf{sense}.(b,w) : b \in \{0,1\}, w \in \{0,\ldots W-1\}\} \\
&\textbf{Outputs} \subset \{\mathsf{WireOut}\} \\
&\textit{enable}(\mathsf{tick}) = 1 \\
&\textit{enable}(\mathsf{sense}.(p,w)) = \mathsf{Level}(w) \neq p \bigwedge \mathsf{Stable}(w,0) \\
&\}
\end{aligned}
$$

Figure: 5.8 The interface of a bus device

We need a priority function priority : $\{0,\ldots,N-1\} \rightarrow \{0,\ldots 2^W-1\}$ which will associate each device with a priority that can be expressed as a W bit number. Priorities should be unique, that is priority(i) = priority(j) should imply that i = j. We can now specify a Futurebus+ system, containing a bus and N devices. There is a single Bus component and components dev.d for each $0 \leqslant d < N$. We insist that the alphabet of the system include a tick symbol, but do not say what other symbols, if any, should be in this alphabet. We insist that each tick of the composite system should force every component to traverse exactly one tick, and that no other bus system transitions should cause time to pass in the components. Thus, time passes uniformly in the system. We also correlate the outputs of devices and the bus. After each instant of time, the output of the bus on pin (w, d) should be communicated to the input of device d on wire w. Similarly, after each tick the outputs of device d on wire w should be communicated to pin (w, d) on the bus.

$bus\_sys(T_{prop}, T_{int}, W, N, pri: \{0, ...N-1\} \rightarrow \{0, ...2^W - 1\})$
{
$(\forall 0 \leqslant i, j \prec N)(pri(i) = pri(j) \leftrightarrow i = j)$
Alphabet $\subset \{tick\}$
Components $= \{dev.0, ..., dev.(N-1), bus\}$
$(\forall d)(\textbf{in}\ dev.d \Box Arb\_dev)(T_{prop}, T_{int}, W, priority(d))$
$(\textbf{in}\ bus \Box Bus\_dev)(T_{prop}, W, N)$
$(\forall c, a)(\textbf{in}\ c)duration(f\_effect(a, c)) = (a = tick)$
$(\forall (w, d))\ ((\textbf{after}\ tick)(\textbf{in}\ bus)Level((w, d)) = (\textbf{in}\ dev.d)WireOut(w))$
$(\forall (w, d))\ ((\textbf{after}\ tick)(\textbf{in}\ Dev.d)Level(w)) = (\textbf{in}\ bus)PinOut((w, d)))$
$\Box\ \textbf{enable}(tick)$
}

Figure: 5.9 Futurebus+ system sketch

We now have a reasonably accurate model of a Futurebus+ system, and can move on to describe the arbitration algorithm. We first assume that the devices can be roughly synchronized around the start of an arbitration cycle. That is, we associate each device with an output, $Enter: \emptyset \rightarrow \{0, 1\}$. A part of the protocol that is not described here makes sure that if one device asserts Enter, then within a certain time period every other device will have committed itself to the competition or to waiting until the next cycle. Let

$$Enter(d) \stackrel{\text{def}}{=} (\textbf{in}\ dev.d)Enter.$$

$$(\exists d)Enter(d) \rightarrow (\forall d)\Diamond_{T_{sync}} \left(\begin{array}{l} Enter(d) \\ \bigvee \neg Enter(d)\ \textbf{until}\ (\forall d)\neg Enter(d) \end{array} \right)$$

Figure: 5.10 Competition synchronization on the FutureBus+

Thus, once one device signals its intention to start a competition cycle, within T_{sync} time units, every device must either be in the competition or must wait until the competition is over.

When device d is not competing, $WireOut(w) = 0$ for all w. When the device is competing for control of the bus, its output on the bus will be a consequence of priority of the device and the current state of the competition. When the competition begins device i sets $WireOut(w) = Bit(w, priority(i))$, where $Bit(k, n)$ is the k^{th} digit of the binary encoding of n. During the competition, a device i

is *losing* at bit w iff for some $w' \geq w$, it detects that another device is asserting a higher value. That is $\mathsf{Losing}(w) \stackrel{\text{def}}{=} (\exists w' \geq w)\mathsf{Bit}(w', \mathsf{priority}(i)) < \mathsf{Level}(w')$. If the device is losing at w, it must set $\mathsf{WireOut}(w) = 0$ until it is no longer losing, or until the competition is over. If the device is losing at w, it must also be losing at $w' < w$, so all lower ordered bits get zeroed. Whenever device d stops losing at wire w, it re-asserts its priority bit. Of course, we have to account for delay in this process, and so need a function $\mathsf{Lost}(w, t)$ which is true iff the device has been losing on w for at least t time units. We also need a competition timer, to keep track of the length of the competition. If $T_{compete}$ is chosen correctly, at the end of $T_{compete}$ time units the competing device which is not losing anywhere must have the highest priority.

$$
\begin{array}{l}
\mathsf{ArbDev}(N, W, T_{prop}, T_{int}, pri) \rightarrow \\
\{ \\
\mathsf{Enter} = 0 \rightarrow (\forall w)\mathsf{WireOut}(w) = 0 \\
\mathsf{Losing}(w) = (\exists w' \geq w)\mathsf{Bit}(w', \mathsf{priority}(i)) < \mathsf{Level}(w') \\
\mathsf{Initial} \rightarrow \mathsf{Lost}(w, t) = 0 \\
(\mathbf{after}\ \mathsf{tick})\mathsf{Lost}(w, t) = \begin{cases} 0 & \mathbf{if} \neg \mathsf{Losing}(w) \\ 1 & \mathbf{if} \mathsf{Losing}(w) \bigwedge t = 0 \\ \mathsf{Lost}(w, t-1) & \mathbf{otherwise} \end{cases} \\
\mathsf{Enter} \bigwedge \neg \mathsf{Lost}(w, T_{int}) \rightarrow \mathsf{WireOut}(w) = \mathsf{Bit}(w, pri) \\
\mathsf{Enter} \bigwedge \mathsf{Lost}(w, T_{int}) \rightarrow \mathsf{WireOut}(w) = 0 \\
\mathsf{Event} = 0 \rightarrow \mathsf{CompTimer} = 0 \\
\mathsf{Event} = 1 \\
\qquad \rightarrow (\mathbf{after}\ a)\mathsf{CompTimer} = \min\{T_{compete}, \mathsf{CompTimer} + (a = \mathsf{tick})\} \\
\mathsf{Winner} = \mathsf{Enter} \bigwedge \mathsf{CompTimer} = T_{compete} \bigwedge (\forall w) \neg \mathsf{Lost}(w, T_{int}) \\
\mathsf{Loser} = \mathsf{Enter} \bigwedge \mathsf{CompTimer} = T_{compete} \bigwedge (\exists w) \neg \mathsf{Lost}(w, T_{int}) \\
\mathsf{Enter} \rightarrow \mathsf{Enter}\ \mathbf{until}\ (\mathsf{Winner} \bigvee \mathsf{Loser}) \\
\{
\end{array}
$$

Figure: 5.11 Device competition algorithm for Futurebus+

There are two correctness properties that need to be verified for this protocol. First, we want to show that there is never more than one device which believes itself to have won.

$$\Box \left(\sum_{d} (\mathsf{in\ dev.d})\mathsf{Winner} \leq 1 \right)$$

Figure: 5.12 Safety condition for Futurebus+

The second correctness condition asserts that the competing device with the greatest priority *will* win the competition within $T_{compete}$ time units. From the synchronization condition, we know that when a competing device has been in the competition for T_{sync} time units, then all devices must have committed themselves to entering the competition or staying out.

$$\left(\begin{array}{l} (\text{in dev.d})\text{CompTimer} \geqslant T_{sync} \\ \bigwedge(\forall d' \neq d)(\neg \text{Enter}(d') \bigvee \text{Priority}(d') < \text{Priority}(d)) \end{array} \right) \rightarrow \Diamond_{T_{compete}} \text{Winner}(d)$$

Figure: 5.13 Liveness condition for Futurebus+

Proving the liveness and safety properties involves using the techniques of the previous section, and techniques that we develop in the next chapter. We will not carry out the proofs here, because they involve close consideration of issues that are not of general interest.

5.4 Summary

We have shown that real-time can be specified within a pure finite state model — without use of external timers. To illustrate the utility of our approach we have considered two, quite complex, examples. The first example, a real-time priority queue, illustrates application of our techniques to a simple scheduling problem, to clarifying complex real-time requirements, and to proving that an implementation meets its specification. The second example, a fragment of the Futurebus+ protocol, illustrates our approach to distributed real-time, and shows how the issues of electrical signal propagation can be addressed.

CHAPTER 6

A FAULT TOLERANT MESSAGE PROTOCOL

In this chapter we specify and prove correctness of a real-world fault-tolerance algorithm. The algorithm, developed by Chang and Maxemchuk [9], guarantees delivery of broadcast messages over a broadcast medium (e.g. an ethernet) in the presence of faults that may cause messages to be lost or only partially delivered. Instead of describing the operation of the algorithm in pseudo-code, as the authors of the algorithm have done, we generate a precise mathematical specification which is amenable to reasonably simple proof techniques. Our analysis clarifies the workings of the algorithm by discarding the complex program scaffolding that obscures the original exposition.

The purpose of the algorithm. Suppose we have a collection of devices site.s : $s \in S$, where S is a set of site identifiers. If these sites are connected by a broadcast network, each site will be able to send messages that are, in general, delivered to all other sites. In an imperfect world, however, messages can be lost, corrupted, or only partially delivered. There are well known methods, e.g., checksums, which allow detection of message corruption and spurious messages. The purpose of the Chang and Maxemchuk algorithm is to overcome the problem of dropped or partially delivered messages. Within a network, collections of *source* sites and collections of *destination* sites are paired to form *reliable broadcast groups*. Sources will transmit messages that are intended for all destination sites belonging to the group. The algorithm allows the sites to detect when a message has been lost, and to force retransmission of the message until it has been successfully delivered.

The obvious algorithm for ensuring reliable broadcast is *positive acknowledgment*; the source repeats the broadcast message until it has received an acknowledgment message from every destination. But positive acknowledgment is time consuming and expensive in terms of message traffic; at best, each broadcast

requires one message from each destination. This expense is especially galling when the network error rate is low, making most of the message transactions pointless. Chang and Maxemchuk exploit the broadcast characteristics of the network to reduce this message traffic considerably. Since the acknowledgment messages are also broadcast, if only one of the destinations sends the source an acknowledgment, all the other destinations can "listen in" on the conversation. By rotating responsibility for acknowledgment among the destinations, the algorithm is usually able to get by with only one acknowledgment per broadcast message. Failures cause additional message traffic, but at a reasonable level. We will show that the algorithm is *safe*, i.e., the sites never incorrectly decide that a message has been delivered.

In order to prove the correctness of the algorithm or even to state the algorithm, we need to develop some model of a network. On the other hand, we want to leave the network as loosely specified as possible, both in order to increase generality, and to allow us to get to the interesting part of the algorithm right aways, without a lengthy pre-amble. To begin with, therefore, our model is rather sketchy. We start with a set S of site identifiers, a set M of messages, and a component site.s for each $s \in S$. We assume that each component can send and receive any message in M.

$$(\textbf{in}\ site.s)((\forall m \in M)(send.n \in \textbf{Alphabet} \bigwedge receive.m \in \textbf{Alphabet})).$$

That is, the alphabet of each site site.s must contain symbols send.m and receive.m for every $m \in M$.

Each site will belong to 0 or more *reliable groups*, with each group consisting of a set of source sites and a set of destinations sites. We let G be a set of group identifiers. Let $SOURCES_g \subset S$ and $DESTS_g \subset S$ name the sources and destinations, respectively, of for group g. The number of destinations in group g, is called the cardinality of the group, and is written $CARD_g$, where $CARD_g = |DESTS_g|$.

Each broadcast message is labeled by: a message type identifier, BCAST, a sequence number provided by the message source, and by both source and group identifiers. Each acknowledgement is labled by a message type identifier ACK, the group identifier, source identifier and sequence number of the broadcast message being acknowledged, and by a timestamp generated by the responsible destination site. The function $tks_g : \mathcal{N}at \rightarrow DESTS_g$ maps each natural number

used as a timestamp to the site identifier for the *tokensite* for that timestamp. Thus, $site.tks_g(t)$ is the destination that is responsible for sending messages with timestamp t. Functions for extracting labeling information from messages are listed in figure6.1

Let $m = (y, id, g, n, t, v)$	
$type(m) = t$	*message type*
$source(m) = id$	*source of data*
$group(m) = g$	*reliable group*
$seq(m) = n$	*broadcast sequence number*
$ts(m) = t$	*the message timestamp*
$sseq(m) = (id, n)$	*the pair of source and sequence number*
$tokensite(m) = tks_{group(m)}(ts(m))$	The site responsible for timestamp $ts(m)$

Figure: 6.1 Messages on the broadcast network

The theorem that we want to prove in this paper states that the algorithm is *safe*. That is when a source site concludes that a broadcast has completed all the destination sites must actually have received the message. Our strategy will be to formalize the algorithm in steps, at each step we state only what we need to advance the proof to the next step. In this way we will clarify our understanding of exactly what makes the algorithm work.

In this example we have made one major simplifying assumption: we assume that sequence numbers and timestamps never need to be rolled over. If timestamps and sequence numbers are represented by 48 bit long bit sequences, and if a broadcast is generated every millisecond for each group, it will take approximately 100 years before we run out of timestamps. In fact, the algorithm will work perfectly well when timestamps need to be recycled, but the proof is slightly more complex. Because of the simplifying assumption, we need only the most elementary information about previous state transitions. Let $Past(a) \stackrel{def}{=} precedes(0, 0, a, 0)$, so that $Past(a)$ is true iff the system has traversed at least one a transition. For example, $Past(receive.m)$ is true iff the system has traversed a receive.m transition.

6.1 The algorithm

The source can conclude that a message has been successfully broadcast if the following conditions have been met.

1. *If the source has sent a broadcast message* m_b *with group* g*;*
2. *And if the source has received an acknowledgment message* m_a *which acknowledges* m_b*;*
3. *And if the source has received an acknowledgment message* m_t *for any message belonging to group* g *(the acknowledgment may acknowledge a message sent by another source), where the timestamp of* m_t *is greater than* CARD_g *plus the timestamp of* m_a*;*

Let's formalize these conditions. We first define a function to test whether or not one message is an acknowledgment for a second.

$$\mathsf{is\_ack}(m_a, m_b) \stackrel{\text{def}}{=} \left(\begin{array}{l} \mathsf{type}(m_a) = \mathsf{ACK} \\ \bigwedge \mathsf{source}(m_a) = \mathsf{source}(m_b) \\ \bigwedge \mathsf{seq}(m_a) = \mathsf{seq}(m_b) \\ \bigwedge \mathsf{type}(m_b) = \mathsf{BCAST} \bigwedge \mathsf{group}(m_b) = \mathsf{group}(m_a) \end{array} \right)$$

Now we can make the safety theorem precise.

Theorem 15:

$$\Box \left(\mathsf{in\ site.s} \left(\begin{array}{l} \mathsf{Past}(\mathsf{send.m}) \\ \bigwedge \mathsf{type}(m) = \mathsf{BCAST} \\ \bigwedge \mathsf{source}(m) = s \\ \bigwedge (\exists m_a, m_t) \left(\begin{array}{l} \mathsf{is\_ack}(m_a, m) \\ \bigwedge \mathsf{Past}(\mathsf{receive.}m_a) \\ \bigwedge \mathsf{Past}(\mathsf{receive.}m_t) \\ \bigwedge \mathsf{type}(m_t) = \mathsf{ACK} \\ \bigwedge \mathsf{group}(m_t) = \mathsf{group}(m) \\ \bigwedge \mathsf{ts}(m_t) > \mathsf{ts}(m_a) + \mathsf{CARD}_g \end{array} \right) \end{array} \right) \right)$$

$$\rightarrow$$

$$(\forall s \in \mathsf{DESTS}_g)(\mathsf{in\ site.s}\ (\mathsf{Past}(\mathsf{receive.m}) \bigvee \mathsf{Past}(\mathsf{send.m})))$$

We specify the algorithm by listing seven properties which must be satisfied by the network in order for the algorithm to work. We then show that the theorem can be deduced from these properties. The goal here is to assume as little as

possible about the global properties of the network, and to try to rely only on local properties of individual sites. We make only two global assumptions. First, we assume that the network does not deliver spurious messages. Second, we assume that in every sequence of CARD_g consecutive timestamps, for every destination d, there is exactly one t, so that $\mathsf{tks}_g(t) = d$. These assumptions seem modest enough. The other properties we require are all local. That is, these properties can be guaranteed by individual sites, no matter how other sites or the network behaves. The basic properties are listed below.

1. *Network safety:* A message m can only be received by a site if the message was previously sent by some site.

$$\begin{array}{l} \mathsf{Initial} \bigwedge (\mathsf{after}\, u)(\mathsf{insite.s})\, \mathsf{Past}(\mathsf{receive.m}) \\ \rightarrow \\ (\exists v \prec u)(v \neq u) \bigwedge (\exists s')(\mathsf{after}\, v)(\mathsf{in\ site.}s')\, \mathsf{Past}(\mathsf{send.m}) \end{array}$$

2. *Tokensite coverage:* Any sequence of CARD_g timestamps should map to the entire set of destinations, i.e.:

$$\{\mathsf{tks}_g(t+i) : 0 \leqslant i < \mathsf{CARD}_g\} = \mathsf{DESTS}_g.$$

3. *Source safety:* A broadcast message m_b with source s can be sent by site s' iff $s = s'$ or s' previously received m_b. The second condition allows rebroadcast.

$$\left[\mathsf{in\ site.s} \left(\begin{array}{l} \mathsf{Past}(\mathsf{send.}m_b) \bigwedge \mathsf{type}(m_b) = \mathsf{BCAST}) \\ \rightarrow \\ \mathsf{source}(m_b) = s \bigvee \mathsf{Past}(\mathsf{receive.}m_b) \end{array} \right)\right]$$

4. *Source uniqueness:* A source cannot send two distinct broadcast messages in the same group with the same sequence number.

$$\left[\mathsf{in\ site.s} \left(\begin{array}{l} \mathsf{Past}(\mathsf{send.m}) \\ \bigwedge \mathsf{Past}(\mathsf{send.m}') \\ \bigwedge \mathsf{group}(\mathsf{m}) = \mathsf{group}(\mathsf{m}') \\ \bigwedge \mathsf{type}(\mathsf{m}) = \mathsf{type}(\mathsf{m}') = \mathsf{BCAST} \\ \bigwedge \mathsf{source}(\mathsf{m}) = \mathsf{source}(\mathsf{m}') \\ \bigwedge \mathsf{seq}(\mathsf{m}) = \mathsf{seq}(\mathsf{m}') \end{array} \right)\right] \rightarrow m = m'$$

5. *Destination safety:* An acknowledgment message m_a can be sent by site s iff $s = \mathsf{tokensite}(m_a)$ or s previously received m_a. The second condition allows rebroadcast.

$$\left(\text{in site.s}\left(\begin{array}{l}\textbf{Past}(\text{send.m}_a)\bigwedge \text{type}(\text{m}_a)=\text{ACK})\\ \rightarrow\\ \text{tokensite}(\text{m}_a)=\text{s}\bigvee\textbf{Past}(\text{receive.m}_a)\end{array}\right)\right)$$

6. *Timestamp uniqueness:* A tokensite cannot send two distinct acknowledgements with the same timestamp.

$$\left[\text{in site.s}\left(\begin{array}{l}\textbf{Past}(\text{send.m})\\ \bigwedge\textbf{Past}(\text{send.m}')\\ \bigwedge\text{group}(\text{m})=\text{group}(\text{m}')\\ \bigwedge\text{type}(\text{m})=\text{type}(\text{m}')=\text{ACK}\\ \bigwedge\text{ts}(\text{m})=\text{ts}(\text{m}')\end{array}\right)\right]\rightarrow \text{m}=\text{m}'$$

7. *Tokensite safety:* A destination cannot send an acknowledgement message with timestamp t, unless it has received a broadcast matching the acknowledgment, and unless for every timestamp t' less than t the destination has seen both an acknowledgment and a matching broadcast.

$$\left[\text{in site.s}\left(\begin{array}{l}\textbf{Past}(\text{send.m})\bigwedge\text{type}(\text{m})=\text{ACK}\\ \rightarrow\\ (\forall\ \text{t}\leqslant\text{ts}(\text{m}))(\exists\text{m}_a,\text{m}_b)\\ \quad\text{group}(\text{m}_a)=\text{group}(\text{m})\\ \quad\bigwedge\text{type}(\text{m}_a)=\text{ACK}\\ \quad\bigwedge\text{type}(\text{m}_b)=\text{BCAST}\\ \quad\bigwedge\text{is\_ack}(\text{m}_a,\text{m}_b)\\ \quad\bigwedge\left(\begin{array}{l}(\text{source}(\text{m}_b)=\text{s}\bigwedge\textbf{Past}(\text{send.m}_b))\\ \bigvee\textbf{Past}(\text{receive.m}_b)\end{array}\right)\\ \quad\bigwedge\left(\begin{array}{l}(\text{tokensite}(\text{m}_a)=\text{s}\bigwedge\textbf{Past}(\text{send.m}_a))\\ \bigvee\textbf{Past}(\text{receive.m}_a)\end{array}\right)\end{array}\right)\right]$$

We can show that the properties listed above guarantee two key derived properties. First, we can show that if a site has received a broadcast message m_b, then it must be the case the the site site.source(m_b) has previously sent m_b. Similarly, we can show that if a site has received an acknowledgment message m_a, it must be the case that site.tokensite(m_a) previously sent m_a.

Lemma 6: *Source origin:* A broadcast message m_b with source s must be sent by site s before it can be received or re-broadcast by any other site.

$$\begin{array}{l}(\text{in site.s}(\textbf{Past}(\text{receive.m})\bigwedge\text{type}(\text{m})=\text{BCAST}))\\ \rightarrow\\ (\text{in site.source}(\text{m}))\,\textbf{Past}(\text{send.m})\end{array}$$

Proof.

{ *Assumption* }

1. Initial
2. (after u)(insite.s) Past(receive.m)

⇒ { *Network safety* }

3. $(\exists v \prec u, s')(v \neq u \bigwedge (\text{after } v)(\text{in site.}s' \ ) \ \text{Past}(\text{send.m}))$

⇒ { *Let* v *be the shortest prefix of* u *satisfying* 3 }
{ *and pick some* s' *satisfying* 3 }

4. (after v)(in site.s') Past(send.m))

⇒ { *lemma premise* }

5. type(m) = BCAST

⇒ { *Source safety* }

6. source(m) = $s' \bigvee$(after v (in site.s' Past))(receive.m)

⇒ {*Network safety* }

source(m) = s'
$\bigvee$
$(\exists v' \prec v, s'')(v' \neq v \bigwedge (\text{after } v' \ (\text{in site.}s'' \ \text{Past}))(\text{send.m}))$

⇒ { *Since* v *is the least prefix satisfying* 3 }

source(m) = $s' \bigvee$ False

⇒ source(m) = s'

⇒ (in site.source(m)) Past(send.m)

Lemma 7: *Timestamp origin:* An acknowledgement message with timestamp t must be sent by the site $\text{tks}_g(t)$ before it can be received by or rebroadcast by any other site.

$$(\text{insite.s})\ \text{Past}(\text{receive.m}) \bigwedge \text{type(m)} = \text{ACK}$$
$$\rightarrow$$
$$(\text{in site.tokensite(m)}\ )\ \text{Past}(\text{send.m})$$

Proof.

1. { *Premise* } Initial
2. { *Premise* }(after u (insite.s)) Past(receive.m)

⇒ { *Network safety* }

3. $(\exists v \prec u, s')(v \neq u \bigwedge (\text{after } v \text{ (in site.}s' \text{ Past}))(\text{send.m}))$

⇒ { *Let* v *be the shortest prefix of* u *satisfying* 3 }
{ *and pick some* s' *satisfying* 3 }

4. (after v (in site.s' Past))(send.m))

⇒ { *lemma premise* }

5. type(m) = ACK

⇒ { *Source safety* }

6. tokensite(m) = $s' \bigvee$ (after v (in site.s' Past))(receive.m)

⇒ {*Network safety* }

source(m) = s'

$\bigvee$

$(\exists v' \prec v, s'')(v' \neq v \bigwedge (\text{after } v' \text{ (in site.}s'' \text{ Past}))(\text{send.m}))$

⇒ { *Since* v *is the least prefix satisfying* 3 }

tokensite(m) = $s' \bigvee$ False

⇒ tokensite(m) = s'

⇒ (in site.tokensite(m)) Past(send.m)

We can now prove the main theorem.

{ *List the theorem hypothesis and fix some constants* }
{ m — *is the broadcast message* }
{ m_a — *is the acknowledgment message for* m }
{ g *is the group of* m }
{ m_T — *is the acknowledgment message with timestamp* }
{ $T > ts(m_a) + CARD_g$ }

1. $(in\,site.s)\,\mathbf{Past}(send.m)$
2. $type(m) = BCAST$
3. $source(m) = s$
4. $is\_ack(m_a, m)$
5. $(in site.s)\,\mathbf{Past}(receive.m_a)$
6. $(in site.s)\,\mathbf{Past}(receive.m_T)$
7. $type(m_T) = ACK$
8. $group(m_T) = group(m) = g$
9. $ts(m_T) = T > ts(m_a) + CARD_g$

{*Pick an arbitrary* $d \in DESTS_g$ }

10. $d \in DEST_g$

$\Rightarrow$ {*By tokenmap coverage* }

11. $(\exists ts(m_a) < t_d \leqslant ts(m_T))tks_g(t_1) = d$

$\Rightarrow$ {*By tokensite origin* }

12. $(\mathbf{in}\ site.tokensite(m_T))\,\mathbf{Past}(send.m_T)$

$\Rightarrow$ {*By tokensite safety* }

13. $$(\exists m_d)\left(\begin{array}{l} ts(m_d) = t_d \\ \bigwedge group(m_d) = group(m_T) = g \\ \bigwedge type(m_d) = ACK \\ \bigwedge \left[\mathbf{in}\ site.tokensite(m_T)\ \left(\begin{array}{l}(\mathbf{Past}(receive.m_d) \\ \bigvee \mathbf{Past}(send.m_d)\end{array}\right)\right] \end{array}\right)$$

$\Rightarrow$ { *By tokensite origin* }
14. (in site.tokensite(m_d)) Past(send.m_d)
$\Rightarrow$ { *since* tokensite(m_d) = d }
15. (in site.d) Past(send.m_d)
$\Rightarrow$ { *by 11* }
16. $t_d \geq ts(m_a)$
$\Rightarrow$ {*By destination safety* }
17. $(\exists m_1, m_2)$ $ts(m_1) = ts(m_a)$
$group(m_1) = group(m_a) = g$
$\bigwedge type(m_1) = ACK$
$\bigwedge type(m_2) = BCAST$
$\bigwedge is\_ack(m_1, m_2)$
$\bigwedge$ (source(m_1) = d $\bigwedge$(in site.d) Past(send.m_2))
$\bigvee$(in site.d) Past(receive.m_2)
$\bigwedge$ (tokensite(m_1) = d $\bigwedge$(in site.d) Past(send.m_1))
$\bigvee$(in site.d) Past(receive.m_2)
$\Rightarrow$ { *By tokensite origin* }
18. (in site.tokensite(m_1)) Past(send.m_1)
$\Rightarrow$ {*From 17* }
$ts(m_a) = ts(m_1) \bigwedge group(m_a) = group(m_1)$
$\Rightarrow$ {*also from 17* }
19. tokensite(m_a) = tokensite(m_1)
$\Rightarrow$ { *By destination uniqueness* }
20. $m_a = m_1$
$\Rightarrow$ {*By source origin* }
21. (in site.source(m_2)) Past(send.m_2)
$\Rightarrow$ {*From 17 and 20,*}
22. $is\_ack(m_a, m) \bigwedge is\_ack(m_a, m_2)$
$\Rightarrow$ {*From 22 and definition of* is_ack }
source(m_2) = source(m)
$\bigwedge$ group(m_2) = group(m_a)
$\bigwedge$ seq(m_2) = seq(m)
$\Rightarrow$ {*Source uniqueness* }
23. $m = m_2$
$\Rightarrow$ {*From 23, and 17* }
24. (in site.d)(Past(receive.m) $\bigvee$ Past(send.m))

6.2 Summary

In this chapter we have shown how a sophisticated, real-world, fault-tolerance algorithm can be clarified and verified in the m.p.r. arithmetic. We formalize the algorithm in a manner which exposes its essential features and strips away much of the complexity of its original exposition. In this chapter we have only

verified the "safety" of the algorithm. There is also a liveness property that is useful to verify and that can be addressed by extending the specification.

CHAPTER 7

CONCLUSION

7.1 Dissertation Summary

This dissertation advances a novel approach to formal specification and verification of systems-level computation. The approach is based on a purely finite state model of computation, and makes use of algebraic and syntactic techniques which have not been previously applied to the problem.

Finite state methods. Our reliance on finite state machines for semantic content is somewhat controversial in present day academic computer science. The finite automaton is widely regarded to be a quaint relic, far too limited for any serious application to the study of computation. In fact, some of the major concerns in formal methods research involve issues that cannot be represented in finite state domains: e.g., non-termination, divergence, and unbounded recursion. But, are these issues really interesting for the study of feasable computation? Do infinite methods really provide us with the proper tools for describing and analyzing the behavior of finite state devices and their programs? The methods described in this work suggest that the answer to these questions is not positive. Of course, infinitary computations are important issues for both mathematics and computer science. If we want to hide the finite and mundane nature of actual computing machines from programmers, then the intended meaning of a program may be best described in an infinitary or unbounded framework. If we want to elucidate the foundations of effective computation, we may not find finite state methods satisfactory. But, when we turn to systems-level computation, engineering considerations are paramount and hiding the concrete nature of computing devices seems counter-productive.

The m.p.r arithmetic demonstrates that a very expressive, abstract, and general computational formalism can be based on a purely finite state model. That the

pumping lemma can be substituted for fixed point arguments in defining "liveness", seems quite significant. In addition, the m.p.r. arithmetic indicates that simply truncating infinitary methods is not the optimal approach for developing finitary methods. The precedence relation is derived from finitary considerations and has no obvious infinitary analog. And the utility of the feedback product of m.p.r. arithmetic should indicate that the theory of finite automata is by no means exhausted as a source of insight into computation.

Arithmetic syntax. The m.p.r. arithmetic has a formal syntax derived from the primitive recursive functions. There are two basic reasons why formal syntax is important when we specify computation. The first reason is that a formal syntax allows us to treat algorithms as mathematical objects. The second reason is that intuition is dangerous in systems engineering. System designs are like stories, if the plot is convincing enough, we may overlook logical inconsistencies. If the syntax is formalized, we can make proofs mechanical and, hopefully, uncover errors that would otherwise be hidden. Thus, the emphasis on formal syntax in verification is well placed. The predicate and propositional logics, however, are not necessarily the best foundation for computational formalisms [55]. The primitive recursive arithmetic is no less formal than the logics, but is, at least to some tastes, clearer and less cryptic than the formal logics. In addition the primitive recursive arithmetic is an arithmetic — based on numbers and number functions. In contrast, formal logics are based on the weaker and more general concepts of "logical deduction". When we study finite state computing devices, the need for this generality is not clear.

7.2 Future Work

7.2.1 Applications of M.p.r. arithmetic

We begin with the following very open question: Can we specify and verify the workings of a real operating system in the m.p.r arithmetic? The answer should be positive, but there is no way to test this short of actually making the effort. Tackling such a large system requires development of some tools and techniques that we did not require for the shorter examples. These include: techniques for translating code into m.p.r specifications, a library of specifications of useful modules, and some automated assistance. There are other,

less ambitious engineering applications that are also of interest. In particular, the m.p.r. arithmetic seems well suited as a programming language for programmable logic devices (PLD). PLD's are, essentially, state machines which can be "programmed" by burning out fuses in a combinatorial circuit. It would be interesting to see if m.p.r exact grammars could be compiled into fuse maps for such devices.

Automation. Verification of a large system would benefit from at least some level of automation. A data base of modules and proofs would certainly make it easier to keep track of progress in a large-scale project. More ambitiously, a proof editor of some sort would increase confidence in proofs and take care of some of the more tedious book-keeping tasks. The proof editor would check each step of a proof to verify that the deduction rule was applied to an expression of the correct form and that the result was appropriate. Proof editors have been developed for temporal logic and there is no obvious barrier to prevent their development for m.p.r. arithmetic. The proof tableau method proposed by [58] is well suited to a proof editor design, and the proof rules for the formalized recursive arithmetic [20] could be provided with a similar method.

Theorem proving. Since there is no decision procedure for $f(x) > 0$ where x is free, a completely automated verifier is impossible for the m.p.r arithmetic. There is, however, reason to believe that partial automation of proofs is possible. The Boyer-Moore theorem prover [23] and the HOL theorem prover [13] provide sophisticated proof assitance for formalisms that are far more computationally complex than the m.p.r arithmetic. The formalized p.r. arithmetic is very promising basis for automated theorem proving. The proof rules proposed by Goodstein are quite powerful, and we conjecture that a theorem prover rooted in arithmetic might have some advantages over those rooted in the more general, and weaker, methods of logic. It is also worth noting that the m.p.r. syntax is quite amenable to restrictions which have appealing consequences in decision complexity. Cobham [12] showed that limiting the strength of allowable recursion in function composition can produce a class of functions which is exactly the class of polynomially evaluable functions. Cook [14] and Buss [8] have also produced some recent interesting results in this area.

Model Checking. Clarke and his colleagues have proposed model checking as an alternative to theorem proving [10, 54]. This technique involves checking the correctness of a modal assertion on a previously defined state machine. It is natural to ask whether this technique can be used to check m.p.r. assertions. An exact grammar is a symbolic representation of a state machine. Can we adapt model checking techniques to use exact grammars in place of explicit state graphs or boolean function representations? If so, then we might be able to avoid some of the thornier problems involved in theorem proving. It may also be possible to combine model checking with theorem proving. The hierarchical nature of m.p.r. specifications may actually simplify some model checking. In this regard the hierarchical graph analysis techniques of Lengauer [36] seem applicable.

The feasability of automating m.p.r proofs depends, to a large degree, on the answers to some theoretical questions that are considered in the next section.

7.2.2 Theoretical investigations

The m.p.r arithmetic takes advantage of only the simplest results in the study of automata products. Both the algebraic properties of feedback automata, and the relationship between the expressiveness of the arithmetic and the complexity of automata which can be defined, seem to merit further investigation.

Algebraic automata. A state machine specified by an exact grammar can be considered to be a tree with aperiodic automata at the leaves and product automata in the interior nodes. It would be interesting to see how such a structure could be factored. That is, if we multiply out the product, is there a simpler way to decompose the structure? Gecseg [17] has shown that the α_i products allow for factoring of automata into "primes", where the prime automata form a finite set. Are the m.p.r feedback structures easier to factor because of their aperiodic base? Can we apply Krohn-Rhodes decomposition [28] to these structures in an efficient manner? The answers to these questions have implications for model checking and possible compilation of specifications into circuits or fuse maps.

Extensions of the functional language. Suppose that we extended the m.p.r. functions by adding an initial function count so that the value of count(a, i)

would be the number of a symbols in the trace *mod* i. M.p.r. expressions would still specify regular languages, but we would now be in the realm of solvable languages rather than aperiodic languages[49]. We have not found a need for this type of expression, but it might be useful for certain systems. It would also result in a larger class of specifiable product form transducers. Alternatively, we could *restrict* the arithmetic. For example, instead of precedes we could allow only a function last: where last(i) is the i^{th} rightmost element of the trace. This restriction would almost certainly limit us to definite languages [51]. Restricting the amount and type of feedback would also change the class of definable structures. There might be a relationship between the size of specifications and the index i on permissible α_i products. This list of such possibilities suggests that the techniques of m.p.r arithmetic may provide new ways of looking at one of the most central and most studied areas in the theory of computation.

BIBLIOGRAPHY

[1] A. Goldberg, u. and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, 1984.

[2] Apt, K., editor. *Logics and Models of Concurrent Systems.* Springer-Verlag, 1985.

[3] Banieqbal, B. and Barringer, H. A study of an extended temporal logic and a temporal fixed point semantics. Technical Report UMCS-86-10-2, University of Manchester, 1986.

[4] Barrington, D. M. Extensions of an idea of mcnaughton. *Mathematical System Theory*, 3(23):to appear, 1990.

[5] Ben-Ari, M., Pnueli, A., and Manna, Z. The temporal logic of branching time. *Acta Informatica*, 20, 1983.

[6] Boute, R. T. On the shortcomings of the axiomatic approach as presently used in computer science. In *Compeuro 88 Systems Design: Concepts Methods, and Tools*, 1988.

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

[8] Buss, S. R. *Bounded Arithmetic.* Studies in Proof Theory. Bibliopolis, Napoli, 1986.

[9] Chang, J. and Maxemchuk, N. Reliable broadcast protocols. *ACM Trans. Computer Systems*, 2(3), august 1984.

[10] Clarke, E. M., A., E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 117–119, 1983.

[11] Clarke, E., O. Grumberb, M., and Browne. Reasoning about networks with many identical finite state processes. Technical Report cmu-cs-86-155, Carnegie-Mellon University, October 1986.

[12] Cobham, A. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science.* North Holland, Amsterdam., 1964.

[13] Cohn, A. The notion of proof in hardware verification. *Journal of Auomtated Reasoning*, 5(2), 1989.

[14] Cook, S. A. Feasibly constructive proofs and the propositional calculus. In *7th A.C.M. Symposium on the theory of computation*, 1975.

[15] de Bakker, J., editor. *Current Trends in Concurrency.* Number 224 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[16] Emerson, E. A. and Halpern, J. Y. Sometimes and 'not never' revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1), January 1983.

[17] Gecseg, F. *Products of Automata.* Monographs in Theoretical Computer Science. Springer Verlag, 1986.

[18] Ginzburg, A. *Algebraic theory of automata.* Academic Press, 1968.

[19] Godel. Pr functions. *FIX*, 21, 1987.

[20] Goodstein, R. L. *Recursive Number Theory.* North Holland, Amsterdam, 1957.

[21] Goodstein, R. L. *Development of Mathematical Logic.* Logic Press, London, 1971.

[22] Gouda, M., G. and Chang, C. Proving liveness for networks of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 8(1), January 1986.

[23] Handbook, A. C. L., editor. *Robert S. Boyer and J. Strother Moore.* Perspectives in Computing. Academic Press, 1988.

[24] Harel, D. Statecharts: A visual formalism for complex systems. Technical report, Weizmann Institute, 1984.

[25] Hartmanis, J. and Stearns, R. E. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, 1966.

[26] Hinman, P. G. *Recursion-Theoretic Hierarchies.* Perspectives in Mathematical Logic. Springer-Verlag, 1978.

[27] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[28] Holcombe, W. *Algebraic Automata Theory.* Cambridge University Press, 1983.

[29] Hopcroft, J. E. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Welsey, Reading MA, 1979.

[30] Kooymans, R., Vytopil, J., and DeRoever, W. Real time programming and asynchronous message passing. In *Proceedings of the 2cd ACM Symp. on Principles of Distributed Programming*, 1983.

[31] Kooymans, R. Specifying message passing systems requires extending temporal logic. In Banieqbal, B., Barringer, H., and Pnueli, A., editors, *Temporal Logic in Specification*, number 398 in LNCS, pages 213–223. Springer-Verlag, 1987.

[32] Kripke, S. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[33] Kroger, F. *Temporal Logic of Programs.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1987.

[34] Ladner, R. E. Application of model theoretic games to discrete linear orders and finite automata. *Information and Control*, 33:281–303, 1977.

[35] Lamport, L. What good is temporal logic? In *Proceedings of the IFIP 9th World Computer Congress*, pages 657–667, Paris, September 1983.

[36] Lengauer, T. Hierarchical planarity testing algorithms. In *13th ICALP 1986*, LNCS. Springer-Verlag, 1986.

[37] Lewis, H. R. and Papadimitrou, C. H. *Elements of the theory of computation.* Prentice-Hall, New Jersey, 1981.

[38] Lynch, N., A. and Merrrit, M. Introduction to the theory of nested transactions. Technical Report TR-367, Laboratory for Computer Science, MIT, 1986.

[39] Milner, R. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[40] Moore, E., editor. *Sequential Machines: Selected Papers*. Addison-Welsey, Reading MA, 1964.

[41] Moser, L. E. and Melliar-Smith, P. M. Formal verification of safety-critical systems. *Software Practice and Experience*, 20(8), 1990.

[42] Moszkowski, B. and Manna, Z. Reasoning in interval temporal logic. Technical Report STAN-CS-83-969, Stanford University, July 1983.

[43] Myhill, J. Finite automata and the represntation of events. Technical Report WADD TR-57-624, Wright Patterson Air Force Base, 1957.

[44] Nerode, A. Linear automaton transformations. In *Proc. AMS*, 1958.

[45] Olderog, E.-R. Process theory: semantics, specifications and verification. In de Bakker, J., editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[46] Ostroff, J. and Wonham, W. Modelling, specifying, and verifying real-time embedded computer systems. In *Symposium on Real-Time Systems*, Dec 1987.

[47] Owicki, S. and Lamport, L. Proving liveness properties of concurrent programs. *ACM TOPLAS*, 4(3), 1982.

[48] Peter, R. *Recursive functions*. Academic Press, 1967.

[49] Pin, J. *Varieties of Formal Languages*. Plenum Press, New York, 1986.

[50] Pnueli, A. Applications of temporal logic to the specification and verification of reactive systems: a survey of curent trends. In de Bakker, J., editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[51] Rabin. Definite automata. *IER Transactions*, 1958.

[52] Ramamritham, K. Correctness of a distributed transaction system. *Information systems*, 8(4), 1983.

[53] Simmons, H. The realm of primitive recursion. *Archives of Mathematical Logic*, 27, 1988.

[54] Sistla, A. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, 1983.

[55] Skolem, T. A critical remark on foundational research. In Fenstad, J. E., editor, *Selected Works in Logic by Th. Skolem*. Universitetsforlagel, Olso, 1970.

[56] Skolem, T. The development of recursive arithmetic. In Fenstad, J. E., editor, *Selected Works in Logic by Th. Skolem*. Universitetsforlagel, Olso, 1970.

[57] Smorynski, C. *Self-Reference and Modal Logic*. Springer-Verlag, 1985.

[58] Smullyan, R. M. *First order logic*. Springer-Verlag, New York, 1968.

[59] Stankovic, J. A. and Ramamritham, K. *Hard Real-Time Systems*, volume 819 of *IEEE Tutorials*. IEEE, 1988.

[60] Vardi, M. and Wolper, P. An automata-theoretic approach to automatic program verification. In *Proceedings of the Symposium on Logic in Computer Science*, June 1986.

[61] Wolper, P. Temporal logic can be more expressive. *Information and Control*, 56(1-2), 1983.