THE ALGEBRAIC FEEDBACK PRODUCT OF AUTOMATA(EXTENDED ABSTRACT).

VICTOR YODAIKEN
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF MASSACHUSETTS
AMHERST, MASSACHUSETTS 01003
YODAIKEN@CS.UMASS.EDU

1. Introduction

Finite state automata provide a clear and faithful mathematical representation of digital devices and programs. Since large-scale digital systems are generally constructed by interconnecting less complex components, the problem of representing such systems in terms of interconnected automata has received a great deal of attention in the computer science literature, e.g., [COGB86, Har84, Kur85]. In this paper we describe an algebraic automata product, called the *feedback product* which can be employed to emulate the intricate control and component interconnection of a wide variety of digital systems[1]. A classical Moore machine or finite state transducer [HU79] provides a mathematical model of a discrete finite state mechanism with output. A *feedback product form transducer* provides a mathematical model of an interconnected system of finite state mechanisms. A product form transducer contains some number of *component* transducers representing sub-systems. Each component transducer is associated with a *feedback function.* When input is provided to the product form machine, each feedback function generates an input sequence for its associated transducer. The generated sequence depends on the input provided to the product machine, the global system state, and the outputs of some or all of the other components. Thus, each system transition corresponds to parallel computations in each of the components. Because the feedback functions can access the outputs of all the components, arbitrary (finite state) synchronization can be modeled.

- Generated sequences can be of differing lengths (or empty), so components may change state at varying rates.

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

[1]The feedback product used here is a variation of the general product described in [Gec86].

- A feedback product form transducer can be "multiplied-out" to obtain a classical transducer, we remain within the domain of finite state machine theory.
- Since components may, themselves, be in product form, we can model multi-layer systems.
- There is no need to make the alphabets of components and the product transducer overlapping or disjoint — the feedback functions, rather than the transition labels, define interactions.

The feedback product provides several advantages over inter-connection techniques previously described in the computer science literature. Among these advantages are: a direct (non-interleaved) model of concurrency, a model for systems containing components that change state at differing rates, and a natural model of encapsulation and information hiding. The feedback product also provides a formal framework for describing concurrent systems *without* making any assumptions about underlying computation environment: e.g., scheduling, storage, or communication mechanisms. We are interested in specification of operating systems, circuits, and real-time controllers. These systems are not implemented within environments that provide uniform, fixed communication primitives, and these systems may contain components which have radically different granularities of state change. We are hesitant to accept, for example, the paradigm of a concurrent algol-like programming language as fundamental. And we are dubious about the prospects of verifying the behavior of systems which implement scheduling, storage and communication mechanisms in terms of a formal framework which assumes the previous existence and character of these mechanisms.

Direct analysis and definition of product form automata would be rather awkward. In preference, we define a modal formal language based on the primitive recursive functions [Pet67, Goo57]. The modal primitive recursive (m.p.r.) functions are evaluated in the implicit context of a product form transducer and *trace* (sequence of transition symbols). The trace is intended to represent the sequence of state transitions which have driven the transducer from its initial state. We never explicitly construct either traces or transducers. Instead, we describe these objects in terms of the values that they confer on m.p.r. functions.

We say a transducer P *satisfies* a m.p.r. expression E iff E is non-zero in the context of (P, w) for every w that does not drive P to an undefined state. In other words, P satisfies E iff E is true in every reachable state of P. Every m.p.r. boolean expression is, thus, a specification of the class of transducers which satisfy it. We can show that some fairly simple extensions to the primitive recursive functions are sufficient to obtain a powerful and expressive language in which to describe product form transducers.

Related formalisms. In some respects, our work is closest to that of Clarke *et al* [CAS83, EH83] and Ostroff [OW87], who have also used modal formalisms to describe finite automata. Our work can be seen as providing an alternative semantics for temporal logic based formalism. We believe that this semantics solves some of the problems that temporal logic presents in terms of composition. Statecharts [Har84] are an extension of state diagrams to allow for description of concurrent and composed systems. While statecharts are quite expressive, they suffer from some of the same limits as traditional state diagrams, e.g., we cannot extend the statechart of a 2 bit shift register to obtain the statechart of a 3 bit shift register. Furthermore the formal semantics of statecharts is exceedingly complex.

Outline. The remainder of this extended abstract is in 2 sections. Section 2 defines product form automata, the m.p.r. functions, and an interpretation. In section 3 we show the expressive range of the m.p.r. functions and develop some proof methods.

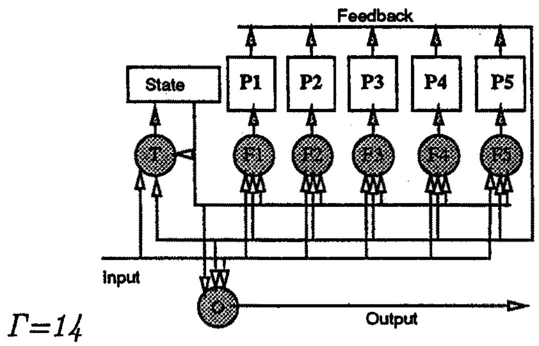
Notational Preliminaries. Much of the body of this paper is concerned with sequences and paths. Let $\langle\rangle$ denote the empty sequence and let juxtaposition denote sequence concatenation: $\langle a_1, \ldots, a_n\rangle\langle b_1, \ldots, b_n\rangle = \langle a_1, .., a_n, b_1, \ldots, b_n\rangle$. Let $a : u \overset{def}{=} \langle a\rangle u$ and let $u :: a \overset{def}{=} u\langle a\rangle$. Finally, let $length(w)$ be the length function.

2. Feedback products and modal functions.

2.1. Product form transducers. 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 [HU79]. The class $\mathbf{P}_{i+1}$ of transducers consists of those transducers containing at least one factor belonging to $\mathbf{P}_l$, and no factors of belonging to $\mathbf{P}_j$ for $j > l$. The class $\mathbf{P}$ is the union over all $\mathbf{P}_i$.

Each product form transducer contains a tuple of *factor* product form transducer and *feedback functions*: $(\phi_1, P_1, \ldots, \phi_r, P_r)$. 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. Note that the transition and output functions can only use the output of the factors, and do not see the internal state of factors.

Figure 2.1. *A product form transducer.*



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

Definition 2.1. *The class of product form transducers*
*The class **P** of product form transducers is the infinite union of the classes $\boldsymbol{P}_i$ for $i \geq 0$. The class $\boldsymbol{P}_0$ consists of all Moore machines. Each class $\boldsymbol{P}_{l+1}$ of product form transducers is the smallest set containing all tuples of the form:*

$$P = (A, O, S, start, \lambda, \delta, (\phi_1, P_1, \ldots, \phi_r, P_r))$$

- *$A = \{1, \ldots n\}$ is a finite transition alphabet, with 0 reserved for use as the null transition that leaves state unchanged,*
- *$O = \{1, \ldots k\}$ is a finite output alphabet,*
- *$S = \{1, \ldots h\}$ is a finite state set (called the top-level state set),*
- *$start \in S$ is a distinguished start state,*
- *Each P_i $(0 < i \leq r)$ is a product form transducer, with $l = max\{l' : (\exists 0 < i \leq r) P_i \in \boldsymbol{P}_{l'}\}$.*
- *$\lambda : S \times O.1 \ldots \times O.r \rightarrow O$,*
- *$\delta : S \times O.1 \ldots \times O.r \times A \rightarrow S$ is a transition function, with $\delta(s, o_1, \ldots, o_r, 0) = s$.*
- *Each $\phi_i : S \times O.1 \ldots \times O.r \times A \rightarrow (A.i)^*$ is a feedback function, with $\phi_i(s, o_1, \ldots, o_r, 0) = \langle\rangle$.*

As the product form transducer accepts input, its internal state changes according to δ, and the outputs of the factors change according to their own transition and output functions and the feedback functions of the product. A *configuration* is a tuple $(s, o_1, \dots, o_r)$ representing a state of the product form transducer where the top-level state is s, and the output of each P_i is o_i. We let $\Delta(P, w)$ represent the configuration reached by following w fro the initial configuration. Let $\psi(P, w, i)$ denote the sequence generated for P_i by ϕ_i when the product transducer follows w from its initial state. Whenever, $\Delta(P_i, \psi(P, w, i))$ is undefined, we want $\Delta(P, w)$ to be also undefined. This means that any trace of a product form transducer will not drive any of the factors into undefined states. The language of traces $\mathcal{L}(P)$ is simply the set of traces that do not drive Δ to an undefined configuration.

2.2. The Modal Primitive Recursive Functions. The class MPR of modal primitive recursive 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 PR of primitive recursive functions [Pet67, Smo85]. Every p.r. function is also a m.p.r. functions. In fact, m.p.r. functions are simply p.r. functions with *hidden* parameters for a product form transducer and its trace. We first define the m.p.r. functions, and then give a partial definition of a map $\rho : MPR \rightarrow PR$ which makes the context dependencies explicit.

Definition 2.2. *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 and the defining rules of the primitive recursive functions.[2].

(1) ***Alphabet:*** $f \overset{def}{=} \textbf{\textit{Alphabet}}$,
(2) ***Output function:*** $f(x) \overset{def}{=} \textbf{\textit{Out}}(x)$,
(3) ***Component names:*** $f \overset{def}{=} \textbf{\textit{Components}}$,
(4) ***Pumping number:*** $f \overset{def}{=} \textbf{\textit{pump\_number}}$,
(5) ***Enabling:*** $f(x) \overset{def}{=} \textbf{\textit{enable}}(x)$,
(6) ***Feedback:*** $f(x, y) \overset{def}{=} \textbf{\textit{effect}}(x, y)$,
(7) ***Path offset:*** $f(y, \vec{x}) \overset{def}{=} (\textbf{\textit{after}}\, y) g(\vec{x})$,
(8) ***Component selection*** $f(y, \vec{x}) \overset{def}{=} (\textbf{\textit{inn}}\, y) g(\vec{x})$.

[2]The form of this presentation is adapted from [Smo85].

Definition 2.3. *A fragment of the evaluation functional, ρ*
Let $P = ((A, O, S, start, \lambda, \delta, (\phi_1, P_1, \ldots \phi_r, P_r))$

If $f \overset{def}{=} \textbf{pump\_number}$

$$(\rho\ f)(P, w, x) \overset{def}{=} size(S) * \Pi_i\, size((\rho\, \textbf{pump\_number})(P_i))$$

If $f(x) \overset{def}{=} \textbf{enable}(x)$

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

If $f(x) \overset{def}{=} \textbf{effect}(x, y)$

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

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

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

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

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

Theorem 2.1. *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. Proceeding 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.*

3. Proof system

Goodstein [Goo57] has described a purely syntactic proof system for p.r. functions, and this system is sound for m.p.r functions by the construction of ρ. We need , however, to provide for modal deductions as well as arithmetic deductions.

3.1. Elementary proof methods. We find it convenient to be able to treat expressions as un-named functions. That is, we might write $(\textbf{inn}\, c)(E \wedge E')$ to force evaluation of the entire expression $t \wedge t'$ within the context of component c. Similarly, we write $(\textbf{after}\, a)(E \wedge E')$ to force the evaluation of the expression $E \wedge E'$ in the state reached via an a transition. We can now list some simple results. The first result is that state independent functions can be "factored" out of the scope of **inn** and **after**.

Theorem 3.1. *Distributive law for modal functionals.*

$$(\boldsymbol{after}\, u)(f(E_1, \ldots, E_n\}) = (\boldsymbol{after}\, u) f((\boldsymbol{after}\, u)E_1, \ldots, (\boldsymbol{after}\, u)E_n)$$

and

$$(\boldsymbol{inn}\, c)(f(E_1, \ldots, E_n)) = (\boldsymbol{inn}\, c) f((\boldsymbol{inn}\, c)E_1, \ldots, (\boldsymbol{inn}\, c)E_n)$$

The second, trivial, result is that primitive recursive functions are state independent.

Theorem 3.2. *Distributive law for non-modal functions. If f is a primitive recursive function then,*

$$(\boldsymbol{after}\, u) f(\vec{x}) = f(\vec{x}) = (\boldsymbol{inn}\, c) f(\vec{x}).$$

A consequence of these two theorems is that many primitive recursive functionals can be factored out of expressions. For example:

$$(\mathbf{after}\, u) \sum f(x) = \sum (\mathbf{after}\, u) f(x).$$

A more interesting theorem allows *inversion* of expressions containing nested modal modifiers.

Theorem 3.3. *Inversion.*

$$(\boldsymbol{after}\, u)(\boldsymbol{inn}\, c) f(\vec{x}) = (\boldsymbol{inn}\, c)(\boldsymbol{after}\, \boldsymbol{effect}(u, c)) f(\vec{x})$$

This theorem states that the result of advancing the entire product state machine to the state reached by following u, and then evaluating $f(\vec{x})$ in the context of component c, is equal to the result of simply advancing component c by the sequence induced by u, and then evaluating $f(x)$.

The "compositional proof rules" that are the staple of process based formal methods are not necessary in m.p.r. arithmetic.

3.2. 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, so that if f meets the restrictions, then $f(\vec{m})$ defines exactly one (minimal) product form automaton. Thus, the construction of f is a proof that the system specified by $f(\vec{m})$ is realizable as a finite state machine. Functions of this restricted form are called *modal grammars.* A modal grammer is a symbolic, and highly compact definition of a family of deterministic finite state machines. Modal grammars provide both a formal basis for exact specifications, and a style of specification that we find intuitive. In this abstract, we do not have space to define modal grammars fully. But a modal grammar is a boolean function defined as a conjunction of clauses which describe the alphabet, component set, feedback, and enabling rules of a product form transducer. It follows from the existence of modal grammars and theorem 2.1 that there is a recursive procedure for deciding $\Box\mathcal{G} \longrightarrow t$ for closed modal grammar $\mathcal{G}$ and arbitrary closed term t.

3.3. Temporal logic style functionals. We let $(\textbf{sometimes}\, u)f(n)$ be true iff $f(n) > 0$ *sometime* during the computation of u. Similarly, we let $(\textbf{always}\, u)f(n)$ be true iff $f(n) > 0$ every state visited by u. We write $v \prec u$ to denote that v is a prefix of u, i.e., $v \prec u \leftrightarrow (\exists z)u = vz$. Note that $u \prec u$ and $\langle\rangle \prec u$.

$$(\textbf{sometimes}\, u)f(\vec{x}) \overset{def}{=} (\exists v \prec u)(\textbf{after}\, v)f(\vec{x}) > 0$$
$$(\textbf{always}\, u)f(\vec{x}) \overset{def}{=} (\forall v \prec u)(\textbf{after}\, v)f(\vec{x}) > 0$$

Let $Paths(i)$ be the set of enabled paths of length i: $Paths(i) = \{u : \textbf{enable}(u) \wedge length(u) = i\}$. We can now state the theorem that makes m.p.r. analogs of temporal logic operators possible.

Theorem 3.4. *There is a total map plen* : $MPR \longrightarrow MPR$ *so that:*

$$\begin{aligned}&(\exists u)\ (\textbf{\textit{after}}\, u)f(\vec{x}) > 0\\ &\qquad \longrightarrow (\exists u \in Paths((plen\ f)(\vec{x})))(\textbf{\textit{sometimes}}\, u)f(\vec{x})\\ &\textit{and}\\ &(\exists u \in Paths((plen\ f)(\vec{x})))(\textbf{\textit{always}}\, u)f(x) > 0\\ &\qquad \leftrightarrow (\forall i)(\exists u in Paths(i))(\textbf{\textit{always}}\, u)f(x)\end{aligned}$$

The implications of theorem 3.4 can be illustrated by considering the concepts of eventuality and henceforth. Suppose that we wish to show that $f(x) > 0$ is inevitable (eventual). Thus, we need to show that there is some i so that no enabled path of length i or more keeps $f(x) = 0$. The second part of the theorem tells us that $f(x)$ is inevitable iff there is no enabled path $u \in Paths((plen\ f)(x))$ so that $(\textbf{always}\, u)(f(x) = 0)$. Similarly, $f(x) > 0$ after every enabled path iff there is no enabled path $u \in Paths((plen\ f)(x))$ so that $(\textbf{sometimes}\, u)(f(x) = 0)$. Thus, we can define constructive versions of the temporal logic operators $\Box$ and $\Diamond$ and their existential counterparts. We can also define a m.p.r. analog of the temporal logic operator ***until***. In the full paper, we show that the axioms of the UB Branching time temporal logic [BAPM83] are valid in m.p.r. arithmetic under our definitions.

REFERENCES

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20, 1983.

[Bou88] R. T. Boute. On the shortcomings of the axiomatic approach as presently used in computer science. In *Compeuro 88 Systems Design: Concepts Methods, and Tools*, 1988.

[CAS83] E. M. Clarke, Emerson A., and A.P. Sistla. 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.

[COGB86] E.M. Clarke, M.C. O. Grumberb, and Browne. Reasoning about networks with many identical finite state processes. Technical Report cmu-cs-86-155, Carnegie-Mellon University, October 1986.

[EH83] E. A. Emerson and J. Y. Halpern. Sometimes and 'not never' revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1), January 1983.

[GC86] G. Gouda, M. and C. Chang. Proving liveness for networks of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 8(1), January 1986.

[Gec86] Ferenc Gecseg. *Products of Automata*. Monographs in Theoretical Computer Science. Springer Verlag, 1986.

[Goo57] R. L. Goodstein. *Recursive Number Theory*. North Holland, Amsterdam, 1957.

[Har84] D. Harel. Statecharts: A visual formalism for complex systems. Technical report, Weizmann Institute, 1984.

[Har87] D. Harel. On the formal semantics of statecharts. In *Proceedings of the Symposium on Logic in Computer Science*, Ithaca, June 1987.

[HU79] John E. Hopcroft and Jeffry D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Welsey, Reading MA, 1979.

[Kro87] F. Kroger. *Temporal Logic of Programs*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1987.

[Kur85] Kurshan. Modelling concurrent processes. In *Proc of Symposia in Applied Mathematics.*, 1985.

[Moo64] E.F. Moore, editor. *Sequential Machines: Selected Papers*. Addison-Welsey, Reading MA, 1964.

[MP79] Z. Manna and A. Pnueli. The modal logic of programs. In *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming*, volume 71 of *Lecture Notes in Computer Science*, New York, 1979. Springer-Verlag.

[OW87] J.S. Ostroff and W.M. Wonham. Modelling, specifying, and verifying real-time embedded computer systems. In *Symposium on Real-Time Systems*, Dec 1987.

[Pet67] Rozsa Peter. *Recursive functions*. Academic Press, 1967.

[Smo85] C. Smorynski. *Self-Reference and Modal Logic*. Springer-Verlag, 1985.