A LOGIC-FREE METHOD FOR MODULAR COMPOSITION OF SPECIFICATIONS

Victor Yodaiken
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-08
February 1, 1990

A Logic-free Method for Modular Composition of Specifications [1]

Victor Yodaiken
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

February 1990

Abstract

Clear mathematical descriptions of large scale computer systems are not possible without extensive use of encapsulation. We argue that standard models of concurrency and composition are too unstructured to support modular composition and verification of systems. We offer an alternative model based on algebraic feedback products of finite state machines. We also describe a technique for concisely specifying complex state machines in terms of state dependent (modal) functions. The product automata model provides a precise interpretation for the formal expressions, and the formal expressions provide an intuitive language for describing multi-layer concurrent digital systems. We develop several examples, showing how specifications of varying levels of abstractness can be composed to specify rather complex systems. The product form of state machines is defined, and we show how formal expressions can be given concrete numeric interpretations.

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

1. Introduction

Modular techniques for the design and implementation of concurrent digital systems are far in advance of the techniques for specifying and verifying concurrent digital systems. Surveys of the current literature [3, 16, 1] demonstrate that, despite some highly sophisticated mathematical approaches, the goal of modular composition of specifications remains very much unattained. The obstacle seems to be inherent in the standard models of concurrency. While there are deep and significant differences among the various trace, net, and logic based computational formalisms, we find that composition of concurrent systems is invariably described in one of two ways. Either the composite system *interleaves* state transitions of the component systems (e.g. [7, 15]), or each composite state transition is associated with an element of the power set of possible component state changes (e.g., [17, 9, 4]). Neither paradigm places much of a restriction on how concurrent components interact. To specify a concrete example of a concurrent system in terms of interleaving or transition power-sets, we must develop a host of supplementary constraints on the nature of the units of computation, on scheduling and on communication. The very generality of these paradigms entails lengthy axiomatization of what would be intuitive and obvious in a more faithful model (c.f., [2]).

In this paper we present a language of formal expressions based on an alternative model of concurrency and composition. We represent composite systems as finite state automata in algebraic *feedback product* form, and specify automata via state dependent (modal) functions. We claim the following:

- A natural, functional style for describing finite state digital systems, for parameterizing specifications, and for modular composition of specifications.
- A precise mathematical interpretation for multi-level specifications in terms of automata and automata products.
- An "open", semantically derived, proof system.

Automata corresponding to substantive systems will tend to be large and complex, and, as we will see in section 3, the algebraic product that we use is not as elementary as string interleavings. For this reason, we have developed a modal (context dependent) arithmetic that provides both a concise notation for specifying and composing automata, and an intuitive proof system for verifying the behavior of automata. Arithmetic expressions are implicitly evaluated in the *context* of the current state of a digital system. The context is represented by a state machine and a sequence of transition symbols we call the *trace*. Traces are sequences of transition symbols which drive the state machine from its initial state without

causing an undefined transition. Expressions may be evaluated in the current context; in the context of a future state obtained by appending additional transition symbols to the trace; or in the context of a component sub-system obtained by replacing the current context with a factor state machine and a "factor" trace.

The formal system is called the *modal primitive recursive* (m.p.r.) arithmetic to denote both its modal (state dependent) nature, and its basis in the primitive recursive arithmetic [6]. The primitive recursive arithmetic is a clear, logic-free formalization of a significant part of integer mathematics [14], containing functions which range over non-negative integers and sequences[1]. We summarize the m.p.r functions below.

- *Component selection.* For each function f, the function $(\mathsf{in}\, c)f$ computes f within the context of a component sub-system c. That is $(\mathsf{in}\, c)f(\vec{x})$ is evaluated by evaluating $f(\vec{x})$ in the context corresponding to factor c of the current state machine.

- *Path offset.* For any state transition a, and any function f, the function $(\mathsf{after}\, a)f$ computes f in the state reached after a state transition a. Let '$\cdot$' denote concatenation of sequences, and let $\langle\rangle$ be the empty sequence. We extend after to sequences by setting $(\mathsf{after}\langle\rangle)f(\vec{x}) = f(\vec{x})$, so that the empty path leads to the current state, and setting $(\mathsf{after}\, u \cdot \langle a\rangle)f(\vec{x}) = (\mathsf{after}\, u)f'(\vec{x})$, where $f'(\vec{x}) \stackrel{\mathsf{def}}{=} (\mathsf{after}\, a)f(\vec{x})$.

- *Enabled transitions.* The m.p.r. boolean function Enable tests the possibility of a future state transition. $\mathsf{Enable}(a) = 1$ iff the a transition is defined from the current state. We extend Enable to paths by letting $\mathsf{Enable}(\langle\rangle) = 1$ and $\mathsf{Enable}(u \cdot \langle a\rangle) = \mathsf{Enable}(u) * (\mathsf{after}\, u)\mathsf{Enable}(a)$, where '*' denotes arithmetic multiplication.

- *Feedback.* Each state change of a composite system induces state transitions in all of the components. The value of $\phi(a, c)$ is the (possibly empty) path induced in component c if the composite system traverses an a transition. If $\nu = \phi(a, c)$, then ν is a sequence of transition symbols over the alphabet of factor c; when the composite system follows a to a new state, c will, in parallel, follow ν to a new state. Since $\phi(a, c)$ may be the empty path, $\langle\rangle$, or a multiple element path, we do not need to require components to change state in lock step. We call ϕ a *feedback* function, because the value of $\phi(a, i)$ may depend on the value of the outputs from one or more of the components: the output of components, thus, influences the input. We extend ϕ to paths by setting $\phi(\langle\rangle, c) = \langle\rangle$, and $\phi(u \cdot \langle a\rangle, c) = \phi(u, c) \cdot (\mathsf{after}\, u)\phi(a, c)$.

- *The precedence function.* The left-right ordering of the trace describes the relative order in which state transitions have happened, with the rightmost symbol in the trace

[1] Formally, some form of *encoding* is used to represent sequences, but this has little practical effect.

representing the most recent state transition. If the current trace contains at least i a symbols, then a pair (a, i) refers to the "i^{th} most recent a" symbol in the trace. If the current trace contains fewer than i a symbols, then (a, i) refers to the null transition. The infix boolean function precedes compares the relative order of two such pairs. The value of (a, i) precedes (b, j) is 1 iff either (a, i) is more recent than (b, j), or (a, i) refers to the null event and (b, j) does not refer to the null event.

- *Everything else.* The primitive recursive functions, and the functions defined by composition and primitive recursion from any m.p.r. functions are also m.p.r. functions.

We will consider an expression to be *true* when it takes on a non-zero value, and *false* otherwise. We say a state machine *satisfies* a specification iff the specification is true in the context of every trace belonging to the state machine. Thus, each n-ary function f, and each appropriate n-vector of constant arguments $\vec{m}$ defines the class of state machines which satisfy $f(\vec{m})$. A detailed specification $f(\vec{m})$ is said to *implement* a more abstract specification $g(\vec{k})$ iff every state machines that satisfies $f(\vec{m})$ also satisfies $g(\vec{k})$.

More abstract (high level) specifications will define large, possibly infinite, classes of satisfying automata. It is often important, however, to describe an algorithm or design in a detailed fashion, and to determine whether or not the design or algorithm is finitely realizable. For example, the specification $enable(a) \wedge (after\ a)f() = f + 1$ defines an unbounded counter that cannot be realized by any finite state machine. A specification $f(\vec{m})$ is called a *modal grammar* if f is defined without use of after, and if f is defined as a conjunction of clauses which describe a finite alphabet, a finite component set, a specification for each element of the component set, and the operation of Enable and ϕ. A modal grammar is *exact* if it defines an empty set of components, or if every component specification is an exact modal grammar. Exact modal grammars are satisfied by exactly one (minimal) finite automaton. Thus, if we can describe a design or algorithm with a modal grammar, we have proved that the algorithm is finitely realizable. A proof of this result, and a formal definition of modal grammars can be found in [18]. There are several examples of modal grammars in the next section.

The remainder of this paper is in three sections: illustrative examples are developed in section 2, the formal semantics is sketched in section 3, and the final section concludes with a summary and comparison to related work.

2. The examples

We introduce the modal arithmetic and style of specification by developing a rather simple example involving shift registers, fifo queues, and latches. The example illustrates our treat-

ment of concurrency and encapsulation: we will show how latches can be used to construct shift registers, and how shift registers can be used as components of larger shift registers and fifos.

2.1. A latch

We begin with a specification of a single bit latch. The alphabet of the latch should be load.0 and load.1, its output function should be $Data() \in \{0,1\}$, and should obey $(after\ load.x)Data() = x$. A modal grammar for the latch can't include the specification $(after\ load.x)Data() = x$ (since modal grammars cannot make use of after), so we use precedes instead. We write $Data() = (load.0, 1)\ precedes\ (load.1, 1)$ to make $Data() = 1$ if the most recent transition was a load.1 transition: if the most recent load.0 preceded the most recent load.1. Note, that in the initial state, $Data() = 0$. The complete modal grammar follows:

Figure 1: Modal grammar for a single bit latch.

$$
\begin{aligned}
cell() \stackrel{def}{=} \{ & \\
& \textbf{Alphabet} = \{load.x : x \in \{0,1\}\} \\
& \wedge\ \textbf{Outputs} = \{Data() \in \{0,1\}\} \\
& \wedge\ Data() = (load.0, 1)\ precedes\ (load.1, 1) \\
& \wedge\ Enable(load.x) = 1 \\
\} &
\end{aligned}
$$

In the sequel we will omit the $\wedge$ symbols connecting clauses of a specification, leaving the conjunction to be implicit. The enabling rules of this system are particularly simple: both possible state transitions are enabled at all times.

To assert that the system obeys cell() in all states, we assert that in the initial state every enabled path leads to a state where $cell() > 0$. We therefore, take a short digression to define a function Initial() which is true only in the initial state, and an operator $\Box$ (always), so that $\Box f(x)$ is non-zero iff $f(x)$ is non-zero in the current state and all future states.

In the initial state, the trace is the empty trace $\langle\rangle$. Thus, in the initial state $(a, 1)precedes\ (b, 1) = 0$ for every a and b.

$$Initial() \stackrel{def}{=} (\forall a, b)((a, 1)\ precedes\ (b, 1) = 0).$$

If $f(\vec{m})$ ever becomes false, then there must be an enabled path u which leads to the state where $f(\vec{m})$ is false. Thus, we define $\Box f(\vec{x})$ as follows.

$$\Box f(\vec{x}) \stackrel{def}{=} (\forall u)(enable(u) \rightarrow (after\ u)f(\vec{x}) > 0).$$

The operator (functional) $\Box$ is inspired by a similar operator found in modal and temporal logics [10, 12, 11].

Thus, Initial() $\rightarrow \Box$cell() specifies that cell() is true in all states.

2.2. A shift register

We turn now to the specification of a shift register. We begin with an abstract specification of the interface and output of such a device, and then define an implementation constructed from latches.

The output of a n bit shift register will be an integer value between 0 and $2^n - 1$. It is sometimes advantageous to be a little more careful in distinguishing between bits and integers, we could define the output to be a vector of binary values, but there is no need for such care here. The alphabet of such a device will consist of symbols Left.0, and Left.1, which represent a single bit shift left, and symbols Right.0, and Right.1, representing single shifts to the right. The high order bits are the rightmost bits, so left.x should divide the current output by 2, and then add $2^{n-1} * x$. Similarly, right.x should multiply the current output by 2, add x and then mod the result by 2^{n-1}.

We will use the same name for the output function of the shift register as we used for the latch — Data. The desired operation and interface of an n bit shift register is given by the specification Shifter(n)

Figure 2: Specification of a Shifter.

$$\text{Shifter}(n) \stackrel{\text{def}}{=} \Box \left(\begin{array}{l} \textbf{Alphabet} \subset \{\text{Left.x}, \text{Right.x} : x \in \{0,1\}\} \\ \wedge\, \textbf{Output} \subset \{\text{Data}()\} \\ \wedge\, (\text{after Left.x})\text{Data}() = (\text{Data}() \textbf{ div } 2) + 2^n * x \\ \wedge\, (\text{after Right.x})\text{Data}() = ((\text{Data}() * 2) + x) \textbf{ mod } 2^{n+1}) \end{array} \right)$$

2.3. A modal grammar for a shift register

There are, of course, any number of possible ways to build a system which implements a shift register. We will develop a straightforward implementation based on n components, latch.0, ..., latch.$(n-1)$, where each component satisfies cell(). The value of Data() will depend on the values of the outputs of the components:

$$\text{Data}() = \sum_{i=0}^{n-1} 2^i * (\textbf{in latch.i})\text{Data}().$$

Note that (in latch.i)Data is the name of a function defined within sub-system latch.i, while Data is the name of a function in the current context. The expression (in latch.i)($\Box$cell)()

is true iff the expression $(\Box \mathsf{cell})()$ is true in the component latch.i. Thus, we specify the behavior of the components, by writing:

$$(\mathbf{in}\ \mathsf{latch.i})(\Box \mathsf{cell})().$$

We consider latch.0 to be the leftmost latch. When the shift register computes Right.x we want latch.0 to load x, and for each $0 < i < n$, we want latch.i to load $(\mathbf{in}\ \mathsf{latch.}(i-1))\mathsf{Data}()$. We formalize this as follows:

$$\phi(\mathsf{Right.x}, \mathsf{latch.i}) = \begin{cases} \langle \mathsf{load.x} \rangle & \mathbf{if}\ i = 0 \\ \langle \mathsf{load.}(\mathbf{in}\ \mathsf{latch.}(i-1))\mathsf{Data}() \rangle & \mathbf{otherwise}. \end{cases}$$

A similar mapping defines the effects of a transition labeled Left.x on the component cells. The complete specification follows.

Figure 3: Modal grammar for a $(n+1)$ bit shift register.

$\mathsf{ShiftRegister}(n) \stackrel{\text{def}}{=} \{$
Alphabet $= \{\mathsf{Left.x}, \mathsf{Right.x} : x \in \{0,1\}\}$
Components $= \{\mathsf{latch.i} : i \in \{0, \ldots, n\}\}$
$(\forall i)(\mathbf{in}\ \mathsf{latch.i})(\Box \mathsf{cell})()$
Outputs $= \{\mathsf{Data}() \in \{0, \ldots 2^n - 1\}\}$
$\mathsf{Data}() = \sum_{i=0}^{n-1} 2^i * (\mathbf{in}\ \mathsf{latch.i})\mathsf{Data}()$

$$\phi(a.x, \mathsf{latch.i}) = \begin{cases} \langle \mathsf{load.x} \rangle & \mathbf{if}\ i = 0 \bigwedge a = \mathsf{Right}; \\ & \mathbf{or\ if}\ i = n \bigwedge a = \mathsf{Left}; \\ \langle \mathsf{load.}(\mathbf{in}\ \mathsf{latch.}(i-1))\mathsf{Data}() \rangle & \mathbf{otherwise\ if}\ a = \mathsf{Right}; \\ \langle \mathsf{load.}(\mathbf{in}\ \mathsf{latch.}(i+1)\mathsf{Data}() \rangle & \mathbf{otherwise}. \end{cases}$$

$\mathsf{Enable}(\mathsf{Left.x}) = 1$
$\mathsf{Enable}(\mathsf{Right.x}) = 1$
$\}$

$\mathsf{ShiftRegister}(n)$ is correct iff:

Figure 4:

$$(\mathsf{Initial}() \bigwedge \Box \mathsf{ShiftRegister}(n)) \rightarrow \Box \mathsf{Shifter}(n).$$

In this paper we will not offer a proof system, but we will mention the two proof rules that are central to the verification of this property. First, we note that in and after only matter to state dependent functions, purely arithmetic functions and function modifiers do not depend on state or context. In particular:

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

We call this rule *invariance*. The second rule, called *modal inversion*, allows us to invert expressions of the form $(\mathbf{after}\ u)(\mathbf{in}\ c)f(\vec{x})$. Note that the value of $(\mathbf{in}\ c)f(\vec{x})$ in the state

reached after u, depends on the path that u induces for component c. This path is given by $\phi(u, c)$. Thus, the *inversion* rule is as follows:

$$\phi(u,c) = v \rightarrow (\mathsf{after}\,u)(\mathbf{in}\,c)f(\vec{x}) = (\mathbf{in}\,c)(\mathsf{after}\,v)f(\vec{x}).$$

More details of the proof system can be found in [18]. In this paper we are more concerned with illustrating composition than with illustrating proof techniques. In the next sub-section we describe a "big" shift register composed of serially connected ShiftRegisterss.

2.4. A composite shift register

Suppose that we want to construct a larger shift register from n bit shift registers. Clearly we can connect the component registers in series.

Figure 5: Modal grammar for a $(m+1)*(n+1)$ bit shift register, constructed from $m+1$ serially connected shift registers.

$$\begin{array}{l}
\mathsf{BigRegister}(m,n) \stackrel{\mathrm{def}}{=} \{ \\
\textbf{Alphabet} = \{\mathsf{Left.x}, \mathsf{Right.x} : x \in \{0,1\}\} \\
\textbf{Components} = \{\mathsf{Reg.i} : i \in \{0,\ldots,m-1\}\} \\
(\forall i)(\mathbf{in}\,\mathsf{Reg.i})(\Box\mathsf{ShiftRegister})(n) \\
\textbf{Outputs} = \{\mathsf{Data}() \in \{0,\ldots 2^{m-n+1}-1\}\} \\
\mathsf{Data}() = \sum_{i=0}^{m-1} 2^i * (\mathbf{in}\,\mathsf{Reg.i})\mathsf{Data}() \\
\phi(a.x, \mathsf{Reg.i}) = \begin{cases}
\langle \mathsf{Right.x} \rangle & \textbf{if } i = 0 \bigwedge a = \mathsf{Right}; \\
\langle \mathsf{Left.x} \rangle & \textbf{if } i = m-1 \bigwedge a = \mathsf{Left}; \\
\langle \mathsf{Right.}(\mathbf{in}\,\mathsf{Reg.}(i-1))\mathsf{Data}() \rangle & \textbf{otherwise if } a = \mathsf{Right}; \\
\langle \mathsf{Left.}(\mathbf{in}\,\mathsf{Reg.}(i+1))\mathsf{Data}() \rangle & \textbf{otherwise.}
\end{cases} \\
\mathsf{Enable}(\langle \mathsf{Left.x} \rangle) = 1 \\
\mathsf{Enable}(\langle \mathsf{Right.x} \rangle) = 1 \\
\}
\end{array}$$

The correctness property of interest for the big shift register will be:

Figure 6:

$$(\mathsf{Initial}() \bigwedge \Box\mathsf{BigRegister}(n,m)) \rightarrow \Box\mathsf{Shifter}(m*n).$$

2.5. A fifo queue

Finally, let's construct a first in first out queue (fifo) by connecting shift registers in parallel. Suppose we want to be able to enqueue integers in the range $X = \{0, \ldots, 2^m - 1\}$ on a queue with maximum length n. The alphabet of the queue should contain symbols enq.x and deq.x

for $x \in X$. The output of the queue should include two boolean signals, Empty(), and Full(), and a data value Front() $\in X$ which reflects the value of the element at the head of the queue.

We construct the fifo from m *data* shift registers which hold the enqueued words, and an additional *control* shift register which keeps track of the head of the queue. Any element of X can be represented as a m length binary string, and we will store each of these bits in a *data* shift register. Let Bit(i, k) denote the i^{th} bit of the binary representation of k, i.e.,

$$\mathsf{Bit}(i,k) \stackrel{\text{def}}{=} \begin{cases} 1 & \textbf{if } k \bmod 2^{i-1} \geqslant 2^i; \\ 0 & \textbf{otherwise.} \end{cases}$$

The data registers will be named reg.0, ..., reg.$(m-1)$. When we enqueue x, data register Reg.i will compute Right.Bit(i, x), storing the i^{th} bit of x. The control register will compute Right.1 at the same time. When we dequeue the head element, the control register will compute Left.0. Thus, the position of the rightmost 1 bit in the control register marks the position of the head of the queue. This position is given by the following function.

$$\mathsf{RightBit}(x) \stackrel{\text{def}}{=} (\max i \leqslant x)\mathsf{Bit}(i,x) > 0.$$

Let l be the value of the control register data output. When $l = 0$ no elements belong to the queue. If $l > 0$ then the element at the head of the queue is given by:

$$\sum_{i=0}^{m-1} 2^i * \mathsf{Bit}(l, (\textbf{in Reg.}i)\mathsf{Data}()).$$

The full fifo specification is given below.

Figure 7: Specification of a m bit wide, n bit deep fifo constructed from $(m+1)$ parallel shift registers.

$$\begin{array}{l}
\mathsf{fifo}(m,n) \stackrel{\text{def}}{=} \{\textbf{Alphabet} = \{\mathsf{deq}, \mathsf{enq}.x, x \in \{0, \ldots 2^m - 1\}\} \\
\textbf{Components} = \{\mathsf{control}, \mathsf{Reg}.i : i \in \{0, \ldots, m-1\}\} \\
(\textbf{in control})(\Box\mathsf{Shifter})(n) \\
(\forall i)(\textbf{in Reg.}i)(\Box\mathsf{Shifter})(n) \\
\textbf{Outputs} = \{\mathsf{Empty}(), \mathsf{Full}() \in \{0,1\}, \mathsf{Front}() \in \{0, \ldots, 2^m - 1\}\} \\
\mathsf{Level}() \stackrel{\text{def}}{=} \mathsf{RightBit}((\textbf{in control})\mathsf{Data}()) \\
\mathsf{Full}() = ((\textbf{in control})\mathsf{Data}() = 0) \\
\mathsf{Empty}() = (\mathsf{Level}() = n-1) \\
\mathsf{Front}() = \sum_{i=0}^{m-1} 2^i * \mathsf{Bit}(\mathsf{Level}(), (\textbf{in Reg.}i)\mathsf{Data}()) \\
\phi(a,c) = \begin{cases} \langle \mathsf{Left}.0 \rangle & \textbf{if } c = \mathsf{control} \bigwedge a = \mathsf{dequeue}; \\ \langle \mathsf{Right}.1 \rangle & \textbf{if } c = \mathsf{control} \bigwedge (\exists x) a = \mathsf{enq}.x; \\ \langle \mathsf{Right}.x \rangle & \textbf{if } c \neq \mathsf{control} \bigwedge a = \mathsf{enqueue}.x; \\ \langle\rangle & \textbf{otherwise.} \end{cases} \\
\mathsf{Enable}(\mathsf{enqueue}.x) = \neg\mathsf{Full}() \\
\mathsf{Enable}(\mathsf{dequeue}) = \neg\mathsf{Empty}() \\
\}
\end{array}$$

The fifo specification is not a modal grammar, because we use the more general Shifter(n) specification for the component registers, rather than the modal grammar ShiftRegister(n). Suppose that the correctness condition of figure 4 is true. Let fifo'(m, n) be a modal grammar obtained by replacing all instances of the expression (□Shifter)(n) in fifo(m, n) with the expression (□ShiftRegister)(n). Then fifo'(n, m) is an exact modal grammar, and:

$$\Box\text{fifo}'(n, m) \rightarrow \Box\text{fifo}(n, m),$$

but the converse proposition is false. The specification BigShifter(n, m) is also a modal grammar. Suppose that the correctness condition of figure 6 is true. Pick k, k' and n so that $k * k' = n$, then let fifo''(m, n) be obtained by replacing all instances of the expression (□Shifter)(n) with (□BigShifter)(k, k'). Then fifo''(n, m) is an exact modal grammar, and

$$\Box\text{fifo}''(n, m) \rightarrow \Box\text{fifo}(n, m),$$

will also be true.

3. Semantics

In this section we associate m.p.r. expressions with precise mathematical semantics in terms of finite state machines. We will define a class of product form automata, so that if P is a product form automaton, and if w is a trace of P, then (P, w) confers a unique integer value on each closed (no-free variables) m.p.r. expression.

A *Moore machine* [8, 13] is a finite state machine with output. The Moore machine consists of a finite alphabet A, a state set Q, a finite output alphabet Y, a transition function $\delta : Q \times A \rightarrow Q$, and an output function $\lambda : Q \rightarrow X$.

$$\mathcal{M} = (A, Q, Y, \delta, \lambda).$$

By convention a *state set* is a finite set Q with a distinguished element $s \in Q$ called the *start state.*

The *traces* of a Moore machine are defined by extending δ to strings. Let $\Delta(q, \langle\rangle) = q$ and let $\Delta(q, \langle a \rangle \cdot u) = \Delta(\delta(q, a), u)$. In the sequel, we abuse notation and write δ for Δ. We write $\delta(q, w) = \perp$ when δ is otherwise undefined on q and w. Thus, w is a trace of M iff $\delta(s, w) \neq \perp$.

We now define a *feedback product* of Moore machines. This product is, essentially, the *general product* described in [5], modified to more closely represent our model of composition of systems. In particular, we have taken care to make sure that only the output of the components is visible to the product. Suppose that $M = (M_1, ...M_n)$, where each $M_i = (A_i, Q_i, Y_i, \delta_i, \lambda_i)$, is a Moore machine. Let A be an alphabet, let Q_{sync} be a state set with

start state $s_{sync} \in Q_{sync}$, and let Y be an output alphabet. We can construct a product from M, Q_{sync}, and Y given the following synchronization functions.

Feedback.
$\Phi : Q_{sync} \times Y_1 \times ... \times Y_n \to (A_1^* \times A_n^*)$,
Synchronizer output.
$\lambda_{sync} : Q_{sync} \times Y_1 \times ... \times Y_n \to X$,
Synchronizer transition.
$\delta_{sync} : Q_{sync} \times Y_1 \times ... \times Y_n \to Q_{sync}$,
Encapsulation.
$\Lambda((q_{sync}, q_1, \ldots, q_n) = (q_{sync}, \lambda_1(q_1), \ldots, \lambda_n(q_n))$.

We denote the feedback product of all these elements as follows:

$$M = \prod_{i=1}^{n} M_i[A, Q_{sync}, Y, \Phi, \lambda_{sync}, \delta_{sync}].$$

The product state machine will have: alphabet A, state set $Q = (Q_{sync} \times Q_1 \ldots \times Q_n)$ with start state $s = (s_{sync}, s_1, \ldots, s_n)$, and output alphabet Y. Let $q = (q_{sync}, q_1, \ldots, q_n) \in Q$, The output function is defined as follows:

$$\lambda(q) = \lambda_{sync}(\psi(q)).$$

Let $\Phi(q, a) = (w_1, ..., w_n)$. Then the transition function is defined as follows:

$$\delta(q, a) = \begin{cases} \perp & \text{if } (\exists 0 < i \leqslant n)\delta_i(q_i, w_i) = \perp; \\ & \text{or if } \delta_{sync}(\psi(q), a) = \perp; \\ (\delta_{sync}(\psi(q), a), \delta_1(q_1, w_1), \ldots, \delta_n(q_n, w_n)) & \text{otherwise.} \end{cases}$$

Let $\Phi_i(q, a) = w_i$ when $\Phi(q, a) = (w_1, \ldots, w_i, \ldots, w_n)$. As with δ we will want to identify Φ with its natural (homomorphic) extension to strings: $\Phi_i(q, \langle\rangle) = \langle\rangle$ and $\Phi_i(q, \langle a \rangle \cdot u) = \langle \Phi_i(q, a) \rangle \cdot \Phi(\delta(q, a), u)$. Because of the way we have defined δ, each state transition induced for each component must be enabled. More formally:

$$\delta(s, u) \neq \perp \to \delta_i(s_i, \Phi_i(s, u)) \neq \perp.$$

Intuitively, we guarantee that state transitions of the product state machine will respect the rules of the factor state machines.

The *semantic structures* for m.p.r. arithmetic are triples $P = (M, C, P)$, where M is a Moore machine, and either $C = P = ()$, or M is a product over C, and P is a tuple containing the semantic structures of the factors of M. When $P = (M, (), ())$, we say that P is an atomic semantic structure: there are no components. Otherwise $P = (M, C, P)$ where: $P = (P_1, \ldots, P_n)$, each $P_i = (M_i, C_i, P_i)$ is a semantic structure, $C = [(M_1, \ldots M_n), A, Q_{sync}, Y, \Phi, \lambda_{sync}, \delta_{sync}]$, and $M = \prod_{i=1}^{n} M_i[A, Q_{sync}, Y, \Phi, \lambda_{sync}, \delta_{sync}]$,

We can now sketch m.p.r. interpretation. Let P be a product form state machine:

$$P = \left(M, (M, A, Q_{sync}, Y, \Phi, \lambda_{sync}, \delta_{sync}), (P_1, \ \ldots \ , P_n)\right).$$

We write $P, w \models f(\vec{m})$ to denote the value of formal expression $f(\vec{m})$ in the context of P and w. The value of $(P, w \models (a, i)$ precedes $(b, j))$ depends only on w, and should be obvious. Similarly $(P, w \models \mathsf{Enable}(u)) = 1$ iff $\delta(s, w \cdot u) \neq \perp$. If f is a primitive recursive function, then $(P, w \models f(\vec{m}) = f(\vec{m})$, the context is ignored. The critical fragments of the interpretation are for after and in.

Figure 8: Interpretation of after and in.

$$(P, w \models (\mathsf{after}\, u) f(\vec{m})) = (P, w \cdot u \models f(\vec{m}))$$
$$(P, w \models (\mathsf{in}\, c) f(\vec{m})) = (P_c, \mathsf{trace}_c(u) \models f(\vec{m}))$$

4. Conclusion

Formal specification of "low level" digital systems, such as operating systems and computer architectures, cannot be based on notions of encapsulation and control derived from programming languages. The purpose of a programming language is to substitute a generic, uniform computational environment for the machine dependent, irregular "low level" environment provided by the operating sysem. But when we design an operating system, we are designing the underlying machine environment, we cannot assume its previous existence. Thus, a formal notion of composition of operating system components cannot be based on some single communication paradigm. In this paper we have described a technique for modular composition of arbitrary finite automata, without underlying conventions about communication and scheduling. Far from being an impediment to composition, we have found the the absence of complex environmental assumptions simplifies composition.

References

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

[2] 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.

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

[4] M. J. "Fischer and R.L. Ladner. Propositional modal logic of programs. In *Proceedings of the 9th ACM Symposium on Theory of Computing.*, 1977.

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

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

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

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

[9] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. Technical Report 86.4, Eindhoven University of Technology, June 1986.

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

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

[12] 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.

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

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

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

[16] F.H. Vogt, editor. *Concurrency 88: International Conference on Concurrency*, Lecture Notes in Computer Science. Springer-Verlag, 1988.

[17] K. Voss, H.J. Genrich, and G Rozenberg, editors. *Concurrency and Nets: Advances in Petri Nets.* Springer-Verlag, 1987.

[18] V. Yodaiken and K. Ramamritham. Axiomatic specification of automata. Technical Report in preparation, University of Massachusetts, 1990.