Characterizing complex state machines in modal logic

Victor Yodaiken
Krithi Ramamritham

COINS Technical Report 87-106
20 October 1987

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

Abstract

A logic for reasoning about finite state automata and products of finite state automata is shown to have much of the expressive power of branching time temporal logics, a flexible notion of concurrency and constructive model definitions. The main ideas of the logic are introduced and several examples are discussed.

1 Introduction

This paper introduces $\mathcal{T}$, a logic of finite state automata (FSA). The logic provides a surprisingly general and expressive methodology for reasoning about computational devices and programs. The paper is divided into three sections. The first section introduces the logic and some useful operators analogous to those found in branching time temporal logics [1,4]. The second section shows how automata can be constructively defined by $\mathcal{T}$ sentences called *grammars* . The third section presents some elementary examples.

Before introducing the methodology, we address the following natural questions; "why, restrict ourselves to finite state systems?" and "why, use a modal logic to reason about FSA's?" The answer to the first question is that the restriction is not nearly as narrow as one might suppose. Digital computers are finite state devices (even when connected in parallel) and algorithms for digital computers cannot utilize infinite storage or computation. The behavior of an algorithm which implements addition will depend on the storage limitations of the machinery which executes the algorithm. The trick is to be able to use the fact that these limits exist, without having to make the specification depend on particular constant values. This brings us to the answer to the second question. The logic allows us to parameterize, abstract and compose FSA's without loosing ourselves in the tedious details of transition functions and state sets. The state sets of interesting systems are too big and too complex to investigate with standard methods. Modal logic, gives us a perfect framework in which to be able to reason about state changes; a framework which permits propositions about both the current state, and the states that can be reached after some sequence of transitions. [1]

[1] An alternative approach to complex finite state automata is taken by Harel [8] and others who introduce structuring to state diagrams. These approaches are not necessarily incompatible, but we feel that there are evident advantages to our, more analytic, method.

1.1 Trace languages

For any finite state automata M, define the trace language $L(M)$ to be the set of finite paths originating at the start state of M. Clearly, trace languages are regular and there is a unique trace language for every minimal automaton. A trace $w \in L(M)$ can be considered to describe a sequential *history* of transitions – the most recent being the rightmost. Thus, a trace defines a state by describing the sequence of events that have driven a finite state machine from its initial state. It will be convenient to have a definition of trace languages that does not directly refer to automata.

Definition. 1 *Let $\mathcal{A}$ be a finite alphabet and $\mathcal{L}$ be a subset of $\mathcal{A}^*$. Then $\mathcal{L}$ is a trace language iff $\mathcal{L}$ is regular and $uv \in \mathcal{L} \rightarrow u \in \mathcal{L}$.*

We incorporate concurrency into the FSA model by using a generalization of *products* of automata [7,10]. A collection of instances of (not necessarily distinct) automata can be considered to comprise a system of concurrent machines. These machines can be combined to generate a "product" machine so that transitions in the product correspond to parallel transitions (or sequences of transitions) in some number of the "component" machines. A product trace language is defined similarly.

Definition. 2 *Let $X = \{\mathcal{L}_1 \ldots \mathcal{L}_n\}$ be a collection of trace languages over alphabets $\mathcal{A}_1 \ldots \mathcal{A}_n$, respectively, and let d be a function. A trace language $\mathcal{L} \subset \mathcal{A}^*$ is a well defined product of X with respect to d iff for all $i \in \{1 \ldots n\}$ and for all $u, v \in \mathcal{A}^*$*

(1) $d(i,u) \in \mathcal{A}_i^*$

(2) $d(i,\langle\rangle) = \langle\rangle$

(3) $u \in \mathcal{L} \rightarrow d(i,u) \in \mathcal{L}_i$

(4) $d(i,u)d(i,v) = d(i,uv)$

A "product" trace language $\mathcal{L}$ is associated with a *decoding* function $d_{\mathcal{L}}$ which maps traces in $\mathcal{L}$ to traces in the components. The 4 constraints on the decoding function are difficult to specify using standard representations of FSA's but turn out to be rather simple in $\mathcal{T}$. The first property makes sure that every sequence of transitions in the product alphabet maps to a sequence in each component alphabet. The second property makes sure that the start state of the product is unique. The third ensures that the components are never driven into undefined states. The fourth makes sure that components do not change state discontinuously.

Essentially, $\mathcal{T}$ is a formal system of arithmetic, extended to allow bounded assertions about traces, paths and components. The choice of the *base* formalization is pretty arbitrary. We require only the most elementary arithmetic to be able to define state machines. Greater expressive power in the base logic is convenient, but makes decision procedures more complex. In a future paper we will discuss decision complexity and verification implications of choosing limited arithmetics, like the bounded arithmetic of [17], the polynomial or linear time arithmetics of [3,5] or more restrictive systems. For this paper we assume the base arithmetic is PRA (Primitive Recursive Arithmetic) [21] which is decidable and gives us the primitive recursive functions. The technical details of the definition of $\mathcal{T}$ can be found in [23]. For now, suffice it to say that we add to the base logic the following symbols; a set symbol $\mathcal{A}$, representing the alphabet $\{a_1, \ldots, a_k\}$, a set symbol C representing the set of components $\{c_1, \ldots, c_n\}$, some new function and predicate symbols $\{+, E, \prec, \lceil, \kappa\}$ and two new connectives $\{\Longrightarrow, \models\}$. We'll use a typewriter font to distinguish symbols in $\mathcal{T}$ that represent transition labels (like $\mathtt{a}, \mathtt{b}, \mathtt{a}', \mathtt{a}_i \ldots$), from other symbols of $\mathcal{T}$. A path $p = \langle \mathtt{a}_{i_1}, \mathtt{a}_{i_2}, \ldots, \mathtt{a}_{i_n} \rangle$ represents a sequence of transitions. An event

$e = (\mathsf{a}, i)$ represents the i^{th} most recent instance of a in the current trace. We often write $\mathsf{a}^{(i)}$ instead of (a, i).

The truth of formulae in $\mathcal{T}$ is relative to current state, represented by a trace, and possible futures, represented by a trace language. There are five kinds of assertions about state in $\mathcal{T}$.

- $+p$ is true iff the sequence of transitions p "can happen" in the current state. By convention, we add a cyclic transition labeled Λ to all states that have no other outgoing transitions. This ensures that there is always some p such that $+p$.
- $E(\mathsf{a}^{(i)})$ is true if any trace leading to the current state must traverse at least i transitions labeled a. Less formally, $E(\mathsf{a}^{(i)})$ is true if a has happened at least i times.
- $\mathsf{a}^{(i)} \prec \mathsf{b}^{(j)}$ is true iff in every trace that leads to the current state, the i^{th} most recent a preceded the j^{th} most recent b. A necessary condition for $\mathsf{a}^{(i)} \prec \mathsf{b}^{(j)}$ to be true is that $E(\mathsf{b}^{(j)})$ be true, an event which has not happened cannot be preceded another event.
- $p \Longrightarrow A$ is true iff $+p$ implies that p leads to a state where A is true.
- $c \models A$ is true iff the component c is in a state where A is true.

If A is true for a given trace w and trace language $\mathcal{L}$ we write $\mathcal{L}, w \models\!\!\!\models A$ (A is satisfied by $\mathcal{L}$ at w). If A is true for every trace in $\mathcal{L}$ then we say A is a theorem of $\mathcal{L}$ and write $\mathcal{L} \models\!\!\!\models A$. The definition of $\models\!\!\!\models$ is pretty straightforward (again, the details are in [23])

$\mathcal{L}, w \models +p$ **iff** $wp \in \mathcal{L}$
$\mathcal{L}, w \models E(\mathtt{a}^{(i)})$ **iff** there are at least i a's in w.
$\mathcal{L}, w \models [\mathtt{a}^{(i)} \prec \mathtt{b}^{(j)}]$ **iff** the i^{th} a (from the right) is to the left of the j^{th} b in w.
$\mathcal{L}, w \models [p \Longrightarrow A]$ **iff** $wp \notin \mathcal{L}$ or $\mathcal{L}, wp \models A$
$\mathcal{L}, w \models [c \models A]$ **iff** $\mathcal{L}_c, d_{\mathcal{L}}(c, w) \models A$

The function symbol κ is used to allow us to make universal assertions such as "all paths lead to states where A is true" without having to quantify over the (infinite) set "all paths". The function symbol $\lceil$ is used to decode the effects of paths on components. The two functions are introduced more precisely in the next two sections.

1.2 Analogs of the temporal operators

Temporal logic [16,11] is a modal logic which allows very intuitively appealing characterizations of computation in terms of what could or will happen in the future. We can very easily define an analog to the temporal operator "$\bigcirc$"

Definition. 3 $\bigcirc A \equiv_{def} \forall \mathtt{a} \in \mathcal{A}[\mathtt{a} \Longrightarrow A]$

Thus $\bigcirc A$ is true iff A must be true in any next state. Note that our convention about Λ makes it certain that $\exists \mathtt{a} \in \mathcal{A}[+\mathtt{a}]$. $\mathcal{T}$ is really a "branching" logic [1] because there may be more than one outgoing path from any state. Because of this $\neg \bigcirc A$ is not equivalent to $\bigcirc \neg A$ and we can define

Definition. 4 $\circledcirc A \equiv_{def} \exists \mathtt{a} \in \mathcal{A}[+\mathtt{a} \wedge \mathtt{a} \Longrightarrow A]$

as a "weak" version of next. The proposition $\circledcirc A$ is true iff there is *some* transition that takes us to a state in which A is true and now we have a duality $\bigcirc \neg A \equiv \neg \circledcirc A$.

A peculiarity of $\mathcal{T}$ is that we allow only *bounded* quantification. This prevents the construction of assertions about non-regular languages and infinite strings, but one might

suppose that such a restriction limits our ability to make universal assertions about future states. Without unbounded quantification over paths, operators found in temporal logics, such as $\Box A$ (A is henceforth true) and $\Diamond A$ (A is eventually true) appear to be undefinable. Fortunately, we can use the pumping properties of regular languages to define bounded versions of these operators. Suppose we wish to determine, for some sentence A, if there is a path p such that $+p \wedge p \Longrightarrow A$. Note that the atomic formulae of $\mathcal{T}$ fall into 4 "sorts". Arithmetic formulae are those not containing any of the new symbols we have added to the base logic. Clearly, the truth of these kind of formulae is unaffected by current state. For arithmetic A we know that $p \Longrightarrow A$ for any path p. That is, the truth of A is not affected by path p. Suppose, on the other hand, we wish to know whether there is any path p which leads to a state where $+p'$ is true. We know that the trace language we are reasoning about is regular, so that there is a minimal FSA, M, which accepts the language. Let m be the number of states in M. Clearly, if there is any such p there must be one of length less than m. Event formulae are those of the form $E(\mathsf{a}^{(i)})$. If there is any possible path p which contains at least one a then there must be such a p of length less than m. So if there is a path which contains at least i a's, there must be such a path of length less than $i \cdot m$.

The difficult case is that of sentences $\mathsf{a}^{(i)} \prec \mathsf{b}^{(j)}$. Note that there must be upper bounds for the value of the terms i and j, because we only have bounded quantification. Let h be the sum of the upper bounds of i and j and let m be as above. We can show, through a pumping argument that there is a path p which leads to a state where $\mathsf{a}^{(i)} \prec \mathsf{b}^{(j)}$ is true, iff there is such a path of length less than $h \cdot m$. Details of the unremarkable proof can be found in [23]. The intuitive content of the proof is merely that because i and j are bounded we only need remember the last i a's and last j b's to determine the truth of the

inequality.

Let $h_{\mathrm{a}}(A)$ be the upper bound of any term i used as a transition "superscript" in A, i.e., in a subformula $E(\mathrm{a}, i)$ or $\mathrm{a}^{(i)} \prec \mathrm{b}^{(j)}$ of A. Let $h(A)$ be the sum of all the h_{a}'s for every a. The upshot of the argument of the preceding paragraphs is that by interpreting κ as m we can make universal assertions about automata using bounded quantification.

Proposition. 1 $\forall p \in \mathcal{A}^{i \leq \kappa \cdot (h(A)+1)}[p \Longrightarrow A] \rightarrow p' \Longrightarrow A$

For notational clarity we will omit the factor $(h(A)+1)$ in such terms (it is always clear from context).

We'll take a quick detour here to define a construct of *interval* temporal logics [14,15,18]. Let $|p|$ denote the length of path p and let pp' denote concatenation. The function $pref$ maps a path p to the set of "prefixes" of p (including p itself and the empty path).

Definition. 5 $pref(p) \equiv_{def} \{p_1 : p_1 \in \mathcal{A}^{i \leq |p|}, \exists p_2 \in \mathcal{A}^{i \leq |p|}[p = p_1 p_2]\}$

Then

Definition. 6 $A \,\mathbf{dur}\, p \equiv_{def} \forall x \in pref(p)[x \Longrightarrow A]$

In words, $A \,\mathbf{dur}\, p$ is true iff all prefixes of p lead to states where A is true. Now we can define analogs for all of the other operators of branching temporal logic.

Definition. 7 $\Box A \equiv_{def} \forall p \in \mathcal{A}^{\kappa}[A \,\mathbf{dur}\, p]$

Definition. 8 $\Diamond A \equiv_{def} \forall p \in \mathcal{A}^{\kappa}\big[+p \rightarrow \neg[(\neg A) \,\mathbf{dur}\, p]\big]$

Thus, $\Box A$ means that A is true in the current state and in all reachable future states, and $\Diamond A$ means that A becomes true in some future state, no matter which path is taken.

Branching time temporal logics offer weak versions of $\Diamond$ and $\Box$ to allow existential assertions about future paths.

Definition. 9

$$\boxdot A \equiv_{def} \exists p \in \mathcal{A}^{\kappa}[+p \bigwedge (A \,\mathbf{dur}\, p)]$$

Definition. 10

$$\diamondsuit A \equiv_{def} \exists p \in \mathcal{A}^{\kappa}[+p \bigwedge \neg((\neg A) \,\mathbf{dur}\, p)]$$

The predicate $\boxdot A$ is true if there is some non-terminating (cyclic) sequence of transitions which keeps A true. The predicate $\diamondsuit A$ is true if there is some reachable future state in which A is true. We can also define the **until** relation.

Definition. 11 A **until** $B \equiv_{def} \forall p \in \mathcal{A}^{\kappa} \forall x \in pref(p)[(\neg B) \,\mathbf{dur}\, x \rightarrow A \,\mathbf{dur}\, x]$

Informally, A **until** B holds if every path that keeps $\neg B$ true, keeps A true too. This version of **until** is sometimes called *weak* because A **until** B does not imply $\Diamond B$. There are many possible alternate definitions of these operators. The choice made here was in order to take advantage of the proof system developed by [1].

Proposition. 2 *The axioms of UB (Unified Branching Time Logic) are valid in $\mathcal{T}$ given definitions 3-11, and the deduction rules of UB are sound in $\mathcal{T}$*

1.3 Concurrency and composition

The function symbol $\lceil$ is used to refer to decoded paths. The term $p\lceil c$ represents the path $d(c,p)$. Our definition of a well-defined product trace language (definition (Def 2)) ensures that $\lceil$ behaves "sensibly". We write $A_x[y]$ to denote the formula obtained by substituting the term y for the free variable x. Thus

Proposition. 3 $(c \models x \Longrightarrow A)_x[p\lceil c] \rightarrow (p \Longrightarrow [c \models A])$

is the valid assertion that if p encodes a path which leads component c to a state where A is true, then p leads the product automata to a state where $c \models A$ is true. The fourth clause in definition (Def 2) implies that $(p\lceil c)(p'\lceil c) = (pp')\lceil c$. From clause (3) of the definition we can see that

Proposition. 4 $+p \rightarrow \forall c \in C\left[(c \models +x)_x[p\lceil c]\ \right]$

Another important consequence of (Def 2) is that theorems about languages carry over to products.

Proposition. 5 *If $\mathcal{L}'$ is the i^{th} component of $\mathcal{L}$ and $\mathcal{L}' \models A$ then $\mathcal{L} \models [i \models A]$.*

For proof suppose for some $u \in \mathcal{L}$ the sentence $[i \models A]$ is false, then $d_{\mathcal{L}}(i,u)$ cannot be in $\mathcal{L}'$ and thus $d_{\mathcal{L}}$ cannot be well defined.

The model of concurrency used here is a byproduct of the method of composition we use. That is, components, c and c' change state concurrently on product transition, a, iff $\mathrm{a}\lceil c \neq \langle\rangle$ and $\mathrm{a}\lceil c' \neq \langle\rangle$. In effect, this method allows components to be synchronized in arbitrary ways. This notion leads to a view of concurrency that is completely independent of any particular methods of communication, clocking or scheduling. Essentially, any model of communication that does not involve unbounded buffering can be described in $\mathcal{T}$. That is because we can arbitrarily synchronize the transitions of component state machines. For example, a CSP [9] style of communication between components can be defined as

$$\mathrm{a}\lceil c = \mathtt{input}_{c',k} \leftrightarrow \mathrm{a}\lceil c' = \mathtt{output}_{c,k}$$

for message k and "processes" c, c'. At this point we'll introduce some notation borrowed from [9] which makes it simpler to handle large alphabets and indexed other sets. Instead

of littering our notation with multiple subscripts, we will adopt the "dot index" notation. Thus the formula above can be rewritten

$$\mathrm{a}\lceil c = \mathtt{input}.(c', k) \leftrightarrow \mathrm{a}\lceil c' = \mathtt{output}.(c, k)$$

Indexing variables will always be constrained to vary over finite sets.

Note that the properties of decoding functions imply that this method of communication is synchronous.

$$(\mathrm{a}\lceil c = \mathtt{input}.(c', k) \bigwedge +\mathrm{a}) \rightarrow [c' \models +\mathtt{output}.(c, k)]$$

2 Models and grammars

The question that naturally arises for some $\mathcal{T}$ specification A is, "is there a trace language (and a decoding function if components are mentioned in A) that makes A true?" In other words "is there a model for A?" There are two well known methods of approaching this question. The first is to define a "proof system" and to show that if this system proves A then there must be a model for A. The second is to try to "synthesize" a model for A, using A as a constraint on the structure of the model. A proof system for $\mathcal{T}$ is described in [23]. In this section we show how aspects of both methods can be used in $\mathcal{T}$. On the one hand we will take advantage of the soundness of modal deduction for simple proofs. On the other hand we will rely a great deal on "model theoretic arguments" using the particularly transparent nature of regular languages and finite state machines. The key to this approach is to show that trace languages can be defined by certain $\mathcal{T}$ sentences called grammars. Grammars are attempts to capture the algorithmic nature of automata by defining under what conditions which transitions will be enabled. The major result of this section is that there is a deterministic procedure for generating trace languages from

grammars.

2.1 Construction of Trace Languages

A *path free* sentence of $\mathcal{T}$ is a sentence in which neither $+$ predicates or the κ function appear (except within the scope of a $\models$). Note that if $\mathcal{L}, w \models\!\!\!\sim A$ for path free A, then the language $\mathcal{L}$ is irrelevant to the truth of A. Path free sentences are assertions about individual traces, components or arithmetic and thus if $d_{\mathcal{L}} = d_{\mathcal{L}'}$, and A is path free, then $(\mathcal{L}, w \models\!\!\!\sim A) \leftrightarrow (\mathcal{L}', w \models\!\!\!\sim A)$. This fact allows us to define trace languages in terms of path free sentences. For path free A we will further abuse notation and write $w \models\!\!\!\sim A$.

A *simple rule* is a sentence $A \bigwedge \neg E(\Lambda^{(1)}) \rightarrow +\mathrm{a}$, where A is path free and contains no $\models$, or $\lceil$ symbols and $\mathrm{a} \neq \Lambda$. Intuitively, a rule of this form says that if A is true and Λ has never happened then a can happen. The purpose of the conjunct $\neg E(\Lambda^{(1)})$ is to make sure that once Λ happens no other transition can be traversed. Since these conjuncts are always present in rules, we will omit them from the text as an abbreviation. Given a rule $R = (A \rightarrow +\mathrm{a})$ we write $Cons(R)$ for a and $Hyp(R)$ for A. If $Hyp(R)$ is true we say R is enabled. A *simple grammar* is a sentence

$$R_1 \bigwedge R_2 \ldots \bigwedge R_n \bigwedge Default$$

where $R_1, \ldots, R_n$ are simple rules and $Default$ is the sentence

$$\Big(\bigwedge^{0<i\leq n} \neg Hyp(R_i) \Big) \rightarrow [\mathrm{a} = \Lambda \leftrightarrow +\mathrm{a}]$$

asserting that if no rule is enabled, only Λ can happen.

Let $\mathcal{G}$ be a grammar. We claim there is exactly one trace language $\mathcal{L}$ such that $\mathcal{L} \models\!\!\!\sim \mathcal{G}$. First we show there is at most one such language by contradiction. Assume that both $\mathcal{L}$ and $\mathcal{L}'$ satisfy $\mathcal{G}$ and that $\mathcal{L} \neq \mathcal{L}'$. Without loss of generality pick some $w \in \mathcal{L}$ so that

$w \notin \mathcal{L}'$. Every prefix of w must be in $\mathcal{L}$ by definition of trace languages and for the same reason λ, the null string, must be in both $\mathcal{L}$ and $\mathcal{L}'$. So let u be the longest prefix of w in both languages. By hypothesis, there must be some a so that $u\mathsf{a} \notin \mathcal{L}'$ and $u\mathsf{a} \in \mathcal{L}$. But this means that $\mathcal{L}, u \models + \mathsf{a}$ is true while $\mathcal{L}', u \models \neg + \mathsf{a}$. For $\mathcal{L}, u \models + \mathsf{a}$ to be true there must be some rule in R in $\mathcal{G}$ so that $\mathcal{L}, u \models \mathit{Hyp}(R)$ and $\mathit{Cons}(R) = \mathsf{a}$. But, $\mathit{Hyp}(R)$ must be path free so $\mathcal{L}', u \models \mathit{Hyp}(R)$ and thus $\mathcal{L}', u \models + \mathsf{a}$ — contradiction.

We prove that there is a trace language that is a model for $\mathcal{G}$ by construction. The construction is as follows:

$$L(\mathcal{G}) = \{\lambda\} \bigcup \{w\mathsf{a} : w \in L(\mathcal{G}), (w \models (\mathcal{G} \rightarrow + \mathsf{a}))\}$$

We need to prove that $L(\mathcal{G})$ is a trace language. The details of this proof are not given here, but $L(\mathcal{G})$ obviously has the trace prefix property ($wu \in \mathcal{L} \rightarrow w \in \mathcal{L}$). The non-trivial part is proving that $L(\mathcal{G})$ is regular. We say that two strings u, w are $\mathcal{G}$ equivalent if for every suffix v the rules of $\mathcal{G}$ that are enabled by uv are the same as those enabled by wv. Let h_{a} be the maximal upper bound of transition exponents $\mathsf{a}^{(i)}$ in $\mathcal{G}$, let h be the sum of all the h_{a}'s and let n be the size of the alphabet. Then every string in $\mathcal{A}^*$ of length greater than n^h is $\mathcal{G}$ equivalent to some string of length less than or equal to n^h. This establishes a finite indexed congruence on $L(\mathcal{G})$ and proves the language is regular.

Proposition. 6 *If $\mathcal{G}$ is a simple grammar then $\mathcal{L} \models \mathcal{G}$ iff $\mathcal{L} = L(\mathcal{G})$*

To define product automata in $\mathcal{T}$ we need to add axiomatizations of components, and the decoding function. Thus *product grammars* also have conjuncts of the form

$$c \models A$$

where we refer to A as the "type" of c, and conjuncts

$$c\lceil a = p$$

defining the decoding function. The rules of product grammars are like those of simple grammars but have an additional *guard* conjunct.

Definition. 12 $Enables(c, z) \equiv_{def} [c \models +z]$

A product rule has the form:

$$A \bigwedge \neg E(\Lambda^{(1)}) \bigwedge \forall c \in C \; Enables(c, \mathsf{a}\lceil c) \rightarrow +\mathsf{a}$$

The condition A must still be path free but may contain component, $\models$ and $\lceil$ symbols and thus path predicates in the scope of $\models$ symbols. The guard conjunct makes sure that the decoding function of the product automata is well defined.

If the types of all the components mentioned in a product grammar $\mathcal{G}$ are themselves grammars then $\mathcal{G}$ is called *determined* . If the tree of component definitions rooted at $\mathcal{G}$ has only *determined* nodes (the leaves must be simple grammars) then $\mathcal{G}$ is said to be *fully determined* and both $L(\mathcal{G})$ and $d_{L(\mathcal{G})}$ can be deterministically constructed.

2.2 Families of languages

Grammars are intended to specify algorithms. Consider the class of grammars $\mathcal{Q} = \{\mathcal{G} : L(\mathcal{G}) \models\!\!\!\models Fifo\}$ where *Fifo* is a $\mathcal{T}$ proposition about fifo queues. This class is defined without reference to the internal structure of the languages or automata. We do not have to assert that there is any structural or algorithmic mapping between the elements of $\mathcal{Q}$ as is common in the algebraic study of automata [10] and computation [2,13]. We only need assert that every element of $\mathcal{Q}$ "behaves like a fifo". Alternatively, if $\mathcal{G}_x$ is a grammar when x is bound, then $\mathcal{G}_x$ defines a family of trace languages. A useful consequence of this is that we can define a implementation parameterized over various resources. For example we can define a grammar $\mathcal{G}_{x,y}$ for a fifo queue with x parameterizing queue length and y

parameterizing the possible elements of the queue. While unbounded inductive proofs are not a part of $\mathcal{T}$ (since we have no unbounded quantification) we are free to use such proofs in algebra and so may be able to show that

$$\forall i > 0, \forall j > 0 \big[L(\mathcal{G}_{x,y}[i,j]) \models\!\!\!\sim Fifo \big]$$

Product grammars that are not fully determined describe partial algorithms. By a partial algorithm we mean an algorithm that makes use of the properties of a components that may not be fully defined. For example, if $\mathcal{G}$ describes an algorithm that makes use of *Fifo* 's there may be a component definition in $\mathcal{G}$ of the form $c \models Fifo$. Suppose we wish to prove that $\mathcal{G} \rightarrow A$. The grammar will not be fully determined because we have not yet given an algorithm for the fifo so we cannot (even if it were practical) generate $L(\mathcal{G})$ and then verify $L(\mathcal{G}) \models\!\!\!\sim A$ What we can do then is try to prove that $\mathcal{H} \in \mathcal{Q}$ implies that the grammar $\mathcal{G}'$ obtained by redefining the type of c to be $\mathcal{H}$ is such that $L(\mathcal{G}') \models\!\!\!\sim A$. Thus we can consider the *types* of components to be parameters for grammars.

3 Some Examples

In this section several elementary examples are developed. The first example is a counter, illustrating the use of a simple grammar, and the integration of arithmetic and state assertions. The second example example is a store and the third example is a "real-time" store. The fourth example is a fifo queue, which for the fifth example we transform into a lifo queue (stack). In the final example we combine the fifo with the store. Before starting the examples we need a few preliminaries.

3.1 Some preliminaries

First, a notional prelude. If $\forall i \in X[\mathsf{a}.i \in \mathcal{A}]$ then we will often refer to the set $\{\mathsf{a}.i : \forall i \in X\}$ by simply omitting the index variable and just writing $\{\mathsf{a}\}$. All quantification will be bounded and so for clarity of notation we will often write $\forall c$ $(\exists c)$ instead of $\forall c \in C$ $(\exists c \in C)$. and likewise for $\forall \mathsf{a}$ $(\exists \mathsf{a})$. We will use a similar trick for sets other than C and $\mathcal{A}$ whenever convenient. A variable *declaration*

$$\mathbf{var}\ i \in X$$

constrains i to vary over X and allows the use of $\forall i$ in place of $\forall i \in X$ in the text. Variable declarations are basically defining axioms. That is, a variable definition introduces a new symbol in terms of previously defined symbols. As a textual convenience we will sometimes treat variable declarations as ordinary (true by definition) propositions. For example, writing $(\mathbf{var}\ i \in X) \wedge \forall i A(i)$ instead of $\forall i \in X(A(i))$. We will use the same trick for defining formulae. Thus $(A \equiv_{def} D) \wedge B$ is an abbreviation (in a loose sense of the word) for B' where B' is the formula obtained from B by replacing each subformula A in B by D.

A very useful function for referring to the "most recent" of some set of transitions can be simply defined. If B is a set of transition symbols

Definition. 13

$$last\{B\} = \begin{cases} \lambda, & \textit{if } \forall \mathsf{a} \in B \neg E(\mathsf{a}^{(1)}); \\ \mathsf{a} : \mathsf{a} \in B \textbf{ and } \forall \mathsf{b} \in B[\mathsf{b}^{(1)} \preceq \mathsf{a}^{(1)}] & \textit{otherwise.} \end{cases}$$

A particularly useful case is where $B = \mathcal{A}$ and we will abbreviate this

Definition. 14 $Last = \mathsf{a} \equiv_{def} last\{\mathcal{A}\} = \mathsf{a}$

There is an obvious theorem about *Last* which will be useful below

Proposition. 7 $\mathtt{a} \Longrightarrow \mathit{Last} = \mathtt{a}$

3.2 5 quick examples

3.2.1 A mod n counter

The first example presented is a grammar for a mod n counter. This grammar is not at all deep, but illustrates some techniques. The grammar is a conjunction of its clauses.

$\mathit{Counter}(n) \equiv_{def}$	**var**: $i \in \{0 \ldots n-1\}$	
	$\mathcal{A} = \{\mathtt{t}.i : \forall i\}$	
	$\mathit{Last} = \lambda \rightarrow +\mathtt{t}.1$	(R1)
	$\mathit{Last} = \mathtt{t}.i \rightarrow +\mathtt{t}.(i+1 \bmod n)$	(R2)

The variable definition and definition of $\mathcal{A}$ are not strictly part of the grammar, but make it more self-contained. Note that a *Default* rule is present in spirit (by the convention of Section 2.1) although it does not appear in the text. Rule 1 (R1) makes sure that in the initial state the counter can traverse a transition labeled t.1 and Rule 2 makes sure that in each subsequent state, the last transition determines the next transition.

3.2.2 A store

Let V be a set of "values" and L be a set of "locations".

$\mathit{Store} \equiv_{def}$	**var** $l \in L$	
	var $v \in V$	
	$\mathcal{A} = \{\mathrm{put}.(l,v) : \forall v, \forall l\}$	
	$\mathit{true} \rightarrow +\mathrm{put}.(l,v)$	(R1)

We can define a predicate to "fetch" values from locations

$$Value(l,v) \equiv_{def} \text{put}.(l,v)^{(1)} \succ last\{\text{put}.(l,v') \ : v' \neq v\}^{(1)}$$

Suppose that the store requires a certain amount of settling time after each assignment. We can add a new transition symbol t to the alphabet to represent the passing of one "tick" and replace rule $(R1)$ with two rules

$\text{t}^{(k)} \succ Last\{\text{put}\}^{(1)} \rightarrow +\text{put}.(l,v)$ (R1′)

$true \rightarrow +\text{t}$ (R2)

which allow put's only after k ticks have passed (where k is some constant) and always allow time to pass. Now suppose that the locations can be partioned into two disjoint sets, a set F of "fast" locations that need a settling time of k ticks, and a set S of "slow" locations that need a settling time of r ticks where $r > k$. We replace rule $(R1')$ with two rules:

$\text{t}^{(k)} \succ Last\{\text{put}\}^{(1)} \wedge l \in F \rightarrow +\text{put}.(l,v)$ (R1.1″)

$\text{t}^{(r)} \succ Last\{\text{put}\}^{(1)} \wedge l \in S \rightarrow +\text{put}.(l,v)$ (R1.2″)

The predicate *value* should also be redefined to reflect a delay while storage settles.

$$Value'(l,v) \equiv_{def} \forall v'[+\text{put}.(l.v')] \bigwedge \text{put}.(l,v)^{(1)} \succ last\{\text{put}.(l,v') \ : v' \neq v\}^{(1)}$$

If l is not "settled" then $Value'(l,v)$ is not true for any v.

3.2.3 A fifo queue

One can consider a queue to be a ordered set from which the element with the highest priority can be removed ("dequeued") and to which new elements may be added ("en-

queued"). In a fifo (first in first out) queue, the member of the queue enqueued first, is the member with the highest priority.

$$
\begin{array}{ll}
fifo \equiv_{def} & \textbf{var } e, e' \in E \\
& \mathcal{A} = \{\text{enqueue}.e, \text{dequeue}.e\} \\
& member(k) \equiv_{def} \text{enqueue}.e^{(1)} \succ \text{dequeue}.e^{(1)} \\
& Better(e, e') \equiv_{def} [\neg member(e') \vee \text{enqueue}.e'^{(1)} \succ \text{enqueue}.e^{(1)}] \\
& First(e) \equiv_{def} member(e) \wedge \forall e' \neq e[Better(e, e')] \\
& \neg member(k) \mapsto +\text{enqueue}.e \qquad \text{(R1)} \\
& First(e) \rightarrow +\text{dequeue}.e \qquad \text{(R2)}
\end{array}
$$

$Member(e)$ is defined to be true if e has been "enqueued" more recently than it has been "dequeued". $Better(e, e')$ is true if e has higher priority than e' — if either e' is not on the queue at all, or e' was "enqueued" after e was enqueued. Element e is the *First* iff e is on the queue and it is better than all other elements. Rule (R1) asserts that is e is not on the queue then e may be enqueued, and rule (R2) asserts that the *First* element may be dequeued. This implementation will only permit an element e to be enqueued if e is not currently a member of the queue. One may argue that this is a peculiar limitation to put on a queue, but in fact all queues suffer from this limitation. If we cannot distinguish e from e' how can we order them? One usually considers the elements of the queue to "name" storage cells in which values that may or may not be unique are stored. In the final example we present a queue of this sort.

3.2.4 A Stack

The grammar given in the previous section may be trivially modified to obtain a grammar for a lifo queue (stack). Change the definition of *Better* by reversing the transition

inequality

$$Better(e, e') \equiv_{def} [\neg member(e') \bigvee \text{enqueue}.e'^{(1)} \prec \text{enqueue}.e^{(1)}]$$

and the grammar as given defines a stack.

3.2.5 Adding storage to the fifo

We now give a grammar for a fifo, where elements in the fifo are "pointers" to data. To do this we let $L = E$, the set of locations equal the set of elements and then use the previously defined $fifo$ and *store* as components in a product grammar.

$fifo2 \equiv_{def}$	$\mathcal{A} = \{\text{push}.v, \text{pop}\}$	
	$f \models fifo$	
	$s \models store$	
	$\text{push}.v \in \{\text{e}.(v, l)\}$	
	$\text{pop} \in \{\text{d}.l\}$	
	$\text{e}.(v, l)[f = \text{enqueue}.l$	
	$\text{e}.(v, l)[s = \text{put}.(l, v)$	
	$\text{d}.l[f = \text{dequeue}.l$	
	$[f \models +\text{enqueue}.l] \rightarrow +\text{e}.(l, v)$	(R1)
	$[f \models +\text{dequeue}.l] \rightarrow +\text{d}.l$	(R2)

This grammar incorporates several novel features. First, we introduce two components f and s, a $fifo$ and a *store*, respectively. Second, we "hide"some of the index variables of the transitions by defining $\text{push}.v$ to be a variable over the transition terms $\text{e}.(l, v)$ and pop to vary over terms $\text{d}.l$. The goal here is to allow use of the queue without explicit reference to locations. The decoding function is defined implicitly in the next 3 clauses.

The value at the top of the queue is the value stored in the location matching the first element of the $fifo$ component.

$$Top(v) \equiv_{def} \exists l\big[[f \models +\text{dequeue}.l] \bigwedge [s \models value(l, v)]\big]$$

4 Conclusions and current research

We have introduced a method for reasoning about and specifying very large, complex finite state systems. We believe that the combination of modal logic, finite state machines and products of automata provides a very powerful mathematical tool for investigating computational systems. The development of our method was motivated by problems in operating systems and computer architecture. In these fields intricate timing dependencies and the layering of components present a barrier to the use of formal methods developed for "higher level" programming. An illustration of this difficulty in the case of "standard" temporal logic can be found in the ambiguity of $\bigcirc$ when previously specified components are combined. $\mathcal{T}$ permits the logical specification of a system to be defined using the same modularization and layering techniques used in actual system design.

$\mathcal{T}$ is still in an early stage of development. Currently, we are investigating several methods of verification and theorem proving, including a Hilbert style deductive proof system, a model theoretic proof system and applicability of automated verification. The intuitive content of finite state automata helps to make syntactic deductions in $\mathcal{T}$ less "abstract" (i.e. more intuitive) than deductions in "standard" temporal logic. We find that it is often convenient in proofs to be able to combine "logical" deductions with model theoretic ones, e.g. to deduce A and $A \rightarrow B$ from $\mathcal{L}$ and some w and then to deduce B via MP. This type of reasoning is more difficult in modal logics where the algebraic structure of models is more complex or less well defined. A second interesting aspect of deduction systems arises from the "selectable" complexity of the base arithmetic logic. Limiting the complexity of the base logic may be useful in limiting the complexity of decision procedures for $\mathcal{T}$. We are also exploring various automated techniques for applicability to $\mathcal{L}$. The work of [4,20] is very similar in intent to our work and clearly model synthesis is appealing in a

FSA based logic. An important question for us is whether "hierarchical" techniques similar to those which have been applied to Hamiltonian circuit problems [19] and VLSI problems [12] can be used to analyize the hierarchy of FSA's described by a product grammar. Such techniques would make the model checking paradigm more feasible for very large state machines.

References

[1] Ben-Ari, M., Pnueli, A. Manna, Z. "The Temporal Logic of Branching Time". *Acta Informatica*, 20(1983).

[2] Browne, M.C., Clarke, E.M., Grumberg, O. Characterizing Kripke Structures in Temporal Logic Technical Report CMU-CS-87-104, Carnegie Mellon University, Jan. 1987.

[3] Buss, Samuel. *Bounded Arithmetic* Studies in Proof Theory, Bibliopolis, 1987.

[4] Clarke, E. M., Emerson A.P., Sistla, A.P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. Proc. 10th Annual Symp. on Principles of Programming Languages, Austin, 1983 pp.117-119.

[5] Cook, S. A. Feasibly Constructive Proofs and the propositional calculus. Seventh ACM Symp. on Theory of Computing. (1975) 83-97.

[6] Emerson, E. A., and Halpern, J. Y. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *JACM* ,33(Jan 86).

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

[8] Harel, David. Statecharts: A Visual Formalism for Complex Systems, Technical Report, Weizmann Institute, 1984.

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

[10] Holcombe, W.M.L. *Algebraic Automata Theory* Cambridge University Press, 1983

[11] Lamport, L. "Sometime" is sometimes "not never"- On the temporal logic of programs. Proceedings, 7th Annual ACM Symposium on Principles of Programming Languages (Las Vegas. Nev., ACM New York. 1980, pp. 174-185.

[12] Lengauer, Thomas. Efficient Solution of connectivity problems on hierarchically defined graphs. Technical Report #24. Reihe Theoretische Informatik, University of Paderborn, June 1985.

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

[14] Moszkowski, B. and Manna, Z Reasoning in Interval Temporal Logic, Stanford Univ. Tech. report STAN-CS-83-969. July 83.

[15] Moskzowski, B. *Reasoning about Digital Circuits* Ph.D. Thesis, Stanford Univ. June 1983.

[16] Manna, Z., and Pnueli, A. The modal logic of programs. Proceedings of the 6th International Colloquium on Automata, Languages, and Programming. Lecture Notes in Computer Science, vol. 71. Springer-Verlag, New York, 1979, pp. 385-408.

[17] Nelson *Predicative Arithmetic* Mathematics Notes, Princeton University , 1986.

[18] Schwartz, R. L., Melliar-Smith, P.M. and Voght, F. An Interval Logic for Higher level Temporal Reasoning: Language Definitions and Examples, Principle of Distributed Computing, August 1983.

[19] Sislenko, A.O. Context-free grammars as a tool for describing polynomial-time subclasses of hard problems. *Information Processing Letters*, volume 14, number 2, April 1982.

[20] Sistla, A.P. *Theoretical Issues in the Design and Verification of Distributed Systems* Ph.D. Thesis, Harvard University 1983.

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

[22] Thomas, W. Classifying regular events in symbolic logic. *J. Computer Syst. Sci.* 25(1982) 360-370.

[23] Yodaiken, Victor. Ramamritham, Krithi. A modal logic for finite automata. Tech. Report. University of MA. (forthcoming)