Regular languages and products of regular languages as Kripke semantics

Victor Yodaiken

COINS Technical Report 89-58
15 June 1989

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

Abstract

We show how regular string languages can be treated as Kripke semantics for a modal logic, and how feedback products of such languages can be used to model concurrency and composition in computational systems.

1. Introduction

In this paper we describe a technique for mathematically modeling the behavior of digital systems. The technique involves treating prefix closed regular string languages as *Kripke structures* [Kripke] for a modal logic, and composing these structures using a relatively sophisticated algebraic product of automata [Gecseg]. These techniques permit us to concisely represent complex digital systems in terms of their behaviors and their construction from (possibly concurrent) sub-systems.

The body of the paper is in three sections. The first section introduces a "local" assertion language consisting of propositions about strings. In the following two sections we extend the language: first to obtain a modal language of propositions about regular string languages, and then to obtain a modal language of propositions about products of regular languages. The formal logics presented here have been concocted for the express purpose of illustrating the method: a far more expressive free-variable modal calculus is described in [Yodaiken].

2. State and relative order of computations

In this section we describe a local assertion language containing propositions about finite strings. We will define a satisfaction relation $\models$ on strings and propositions so that $w \models p$ when p is a true proposition about w. Each proposition p defines a congruence relation $\sim_p$ on the set of all finite strings $\mathcal{A}^*$, where $w \sim_p u$ is true if and only if:

$$(\forall v_1, v_2 \in \mathcal{A}^*)(v_1 w v_2 \models p) \Leftrightarrow (v_1 u v_2 \models p)$$

Let $[w]_p = \{u : u \sim_p w\}$ and let $\text{States}_p = \{[w]_p : w \in \mathcal{A}^*\}$. The key property of our language of propositions is that for each proposition p, States_p is finite. In fact, any local assertion language with this property could be used in the development of our modal logic.

Suppose we record the computations of a system in a string $w = \langle a_1 \dots a_n \rangle$. The left to right ordering of w reflects the temporal ordering of events — the "most recent" computation is named by a_n, and the i^{th} most recent computation is recorded by $a_{n-(i-1)}$. We describe state through assertions about this relative ordering. We write $P(\alpha, i, \beta, j)$

to assert that "at least j β's have occurred since the i^{th} most recent α occurred." For example, the assertion $P(tick, 10, request, 1)$, is true iff "less than 10 *ticks* (of some clock) have occurred since the most recent *request*." The proposition $P(set, 1, clear, 1)$ asserts that a one bit memory cell is *clear* — the most recent *clear* came after the most recent *set*.

Definition 1: If $\mathcal{A}$ is a finite alphabet, the assertion language $\mathcal{P}(\mathcal{A})$ is the propositional closure of:

$$\{P(\alpha, i, \beta, j) : \alpha, \beta \in \mathcal{A}, i, j > 0\}$$

To interpret the propositions of $\mathcal{P}(\mathcal{A})$ we need to map pairs (α, i) to positions in a string. Given a string $w = \langle a_1, \ldots a_n \rangle$ let $|w| = n$ denote the length of w, and $|w|_\alpha$ denote the number of α symbols in w. If $|w|_\alpha \geq i$ we let $pos(\alpha, i, w)$ be the unique k so that $a_k = \alpha$ and $|\langle a_k \ldots a_n \rangle|_\alpha = i$ If $|w|_\alpha < i$ we let $pos(\alpha, i, w) = 0$. Thus, $pos(\alpha, i, w) < pos(\beta, j, w)$ if and only if the j^{th} most recent β is more recent than the i^{th} most recent α.

We can now define $\models$.

Definition 2: For any $w \in \mathcal{A}^*$ and proposition $p \equiv precedes(a, i, b, j)$, say that w *satisfies* p if and only if $(pos(\alpha, i, w) < pos(\beta, j, w))$.

We write $w \models p$ whenever w satisfies p. The definition of $\models$ is completed through the usual extensions to $p \wedge q$, and $\neg p$.

Take $Depth(\alpha, p)$ to be the maximum i so that $P(\alpha, i, \beta, j)$ or $P(\beta, j, \alpha, i)$ is a subformula of p. It is easily seen that for any w, for which $|w|_\alpha > Depth(p, \alpha)$, there exists a u such that $|u|_\alpha \leq Depth(p, \alpha)$ and $w \sim_p u$. Let $Depth(p)$ be $\sum_{\alpha \in \mathcal{A}} Depth(\alpha, p)$. It follows that for any w with $|w| > Depth(p)$ there must be a u such that $|u| \leq Depth(p)$ and $w \sim_p u$.

Theorem 1: . For any $p \in \mathcal{P}(\mathcal{A})$ the set $States_p$ is finite. In fact the cardinality of $States_p$ is bounded by $n^{Depth(p)}$ where n is the cardinality of $\mathcal{A}$.

At this point it would be possible to speak of the regular languages definable in $P(\mathcal{A})$ in the standard manner, e.g., $\{w : w \models p\}$ (cf. [Pin] (pp. 118-119) and [Ladner]). But, our interest is in describing behaviors and algorithms. We find it natural to describe algorithms by defining the state properties which enable or disable computations. Thus, in the next section we develop a formal language in which the state propositions of $P(\mathcal{A})$ are employed as guards on computations.

3. Unitary systems

We define a modal propositional logic $\mathcal{L}(\mathcal{A})$ as follows:

Definition 3: $\mathcal{L}(\mathcal{A})$ is the smallest set such that:

1) If $p \in P(\mathcal{A})$ then p belongs to $\mathcal{L}(\mathcal{A})$,

2) If $\alpha \in \mathcal{A}$ then $E(\alpha)$ belongs to $\mathcal{L}(\mathcal{A})$,

3) If p belongs to $\mathcal{L}(\mathcal{A})$ and $u \in \mathcal{A}^*$ then (**after** u)p belongs to $\mathcal{L}(\mathcal{A})$,

4) If p and q belong to $\mathcal{L}(\mathcal{A})$ then $p \wedge Q$ belongs to $\mathcal{L}(\mathcal{A})$,

5) If p belongs to $\mathcal{L}(\mathcal{A})$ then $\neg p$ belongs to $\mathcal{L}(\mathcal{A})$.

Intuitively, $E(\alpha)$ asserts that α is *enabled* (can happen) and (**after** u)p asserts that in the state reached by following u from the current state p will hold. For example, $E(time\_out)$ asserts that a *time_out* computation can happen. To assert that the computation *time out* can only occur whenever at least 10 *ticks* have passed since the most recent request we write:

$$P(request, 1, tick, 10) \leftrightarrow E(time\_out)$$

To assert that after a *request* and 2 *ticks* a *time_out* can happen we write (**after** $\langle request, tick, tick \rangle$)$E(time\_out)$.

A prefix closed regular language is a regular language $\mathbf{L}$ with the property that $w\alpha \in \mathbf{L} \rightarrow w \in \mathbf{L}$. The semantic structures for $\mathcal{L}(\mathcal{A})$ are the prefix closed regular languages over alphabet $\mathcal{A}$. Prefix closure reflects our use of strings as computation histories — we cannot

permit discontinuous histories. We write $(\mathbf{L}, w \models' p)$ to denote the truth of proposition p on string w in language $\mathbf{L}$.

Definition 4: A pair $(\mathbf{L}, w)$ *satisfies* a $\mathcal{L}(\mathcal{A})$ proposition p, ($(\mathbf{L}, w) \models' p$)) iff:

1) $p \in \mathcal{P}(\mathcal{A})$ and $w \models p$,

or

2) $p \equiv E(\alpha)$ and $w\langle\alpha\rangle \in \mathbf{L}$,

or

3) $p \equiv (\textbf{after } u)q$ and $(\mathbf{L}, wu)$ satisfies q,

or

4) p is a conjunction or a negation, and the usual conditions apply.

Define a *rule* to be an assertion $p \leftrightarrow E(\alpha)$ where $p \in \mathcal{P}(\mathcal{A})$. We write R_α to denote a rule $p_\alpha \leftrightarrow E(\alpha)$. Define a *unitary grammar* to be a conjunction $\bigwedge_{(\alpha \in \mathcal{A})} R_\alpha$. Thus, a unitary grammar is a set of rules governing every computation named by $\mathcal{A}$.

Theorem 2: . For any unitary grammar $\mathcal{G}$, there is exactly one regular language $\mathbf{L} \subset \mathcal{A}^*$ so that $(\forall w \in \mathbf{L})(\mathbf{L}, w \models' \mathcal{G})$.

Proof. We show that there is a language $\mathbf{L}$ satisfying $\mathcal{G}$ by constructing a state machine M which accepts $\mathbf{L}$. Let p be the conjunction of all the left hand sides of the rules of $\mathcal{G}$, $p = \bigwedge_{\alpha \in \mathcal{A}} p_\alpha$. Let the state set S of $\mathbf{M}$ consist of the congruence classes of A^* defined by p, i.e., let $S = \text{States}_p$. By Theorem 1, S will be a finite set. Define a transition function $\delta : S \times \mathcal{A} \rightarrow S$ so that $\delta([w]_p, \alpha) = [w\alpha]_p$ if and only if $w \models p_\alpha$. Let $\delta([w]_p, \alpha)$ be undefined otherwise. The state machine with state set S, transition function δ, initial state $[\langle\rangle]_p$, and with every state an accepting state, will accept a prefix closed regular language satisfying $\mathcal{G}$.

We show that any such language is unique by contradiction. Assume that there are two such languages $\mathbf{L}, \mathbf{L}'$. Since these languages are prefix closed they must share at least one string, e.g., $\langle\rangle$ the empty string. Without loss of generality assume that there is a string w belonging to both languages and an α so that $w\alpha \in \mathbf{L}$ and $w\alpha \notin \mathbf{L}'$ (if there is no such string $\mathbf{L} = \mathbf{L}'$). Take the rule $p_\alpha \leftrightarrow E(\alpha)$ from $\mathcal{G}$ and note that either $\neg(\mathbf{L}, w \models' p_\alpha \leftrightarrow E(\alpha))$ or $\neg(\mathbf{L}', w \models' p_\alpha \leftrightarrow E(\alpha))$ thus contradicting the premise.

4. Composite systems

Unitary grammars cannot define all prefix closed regular languages. In fact, unitary grammars can only define star-free prefix closed regular languages [Pin] (pp. 87-93). More to the point, unitary grammars are not adequate for describing complex computational systems because they are not compositional — they do not allow us to construct a complex specification from specifications of component parts. We wish to extend the model and logic to provide compositionality, without making assumptions about how components may be interconnected. In order to accomplish this we turn to the theory of algebraic products of automata. The intuitive idea here is that computations in a composite system can be mapped to concurrent computations occurring in the component systems. Thus, we consider a composite system to be a tree, with the leaves representing "unitary" components (components that do not themselves contain components). A computation of the top level system is reflected by concurrent computations by some or all of the components below. In this paper we assume that these trees are always finite.

Since the alphabets of components will not necessarily coincide, we need to be able to identify each component with an appropriate language of propositions. We will also need to be able to name all the components of each system in the tree.

Definition 5: A *component list* is either an empty list, (), or a list, $((\mathcal{A}_1, C_1)\ldots(\mathcal{A}_n, C_n))$, where $\mathcal{A}_i$ is the alphabet of the i^{th} component and C_i is the component list of the i^{th} component.

We are now in a position to define a propositional language $\mathcal{L}(\mathcal{A}, C)$ for alphabet $\mathcal{A}$ and component list C.

Definition 6: For a finite alphabet $\mathcal{A}$ and component list $C = ((\mathcal{A}_1, C_1)\ldots(\mathcal{A}_n, C_n))$, the language $\mathcal{L}(\mathcal{A}, C)$ is the smallest set such that:

1) If p belongs to $\mathcal{L}(\mathcal{A})$ then p belongs to $\mathcal{L}(\mathcal{A}, C)$,

2) If p belongs to $\mathcal{L}(\mathcal{A}_c, C_c)$, then (in c)p belongs to $\mathcal{L}(\mathcal{A}, C)$,

3) If $0 < c \leq n$ and $v \in \mathcal{A}_c^*$ and $\alpha \in \mathcal{A}$, then $D(\alpha, c, y)$ belongs to $\mathcal{L}(\mathcal{A}, C)$,

4) The usual rules apply to conjunctions and negations.

A proposition $D(\alpha, c, y)$ is true in the current state iff the computation of α will cause the the component c to compute the sequence of computations y. An assertion (**in** c)p is true iff p is true in the component system named c. For example, $D(Time\_Out, Dev1, \langle raise\_wire1, drop\_wire2\rangle)$ asserts that when the composite system times out, the component $Dev1$ will compute $\langle raise\_wire1, drop\_wire2\rangle$. The assertion (**in** $Dev1$)$requesting \leftrightarrow$ (**in** $Dev2$)$\neg requesting$ is true iff the two components $Dev1$ and $Dev2$ cannot both *request* at the same instant.

Definition 7: The *semantic structures* for propositions of $\mathcal{L}(\mathcal{A}, C)$, where $C = ((\mathcal{A}_1, C_1) \ldots (\mathcal{A}_n, C_n))$, are all $2n + 1$ tuples

$$M = (\mathbf{L}, M_1, \Phi_1, \ldots M_n, \Phi_n)$$

where:

1) $\mathbf{L}$ is a prefix closed regular language over $\mathcal{A}$ called the language of M,

2) Each M_c is a semantic structure for $\mathcal{L}(\mathcal{A}_c, C_c)$,

3) Each Φ_c is a function: $\Phi_c : \mathcal{A}^* \times \mathcal{A} \rightarrow \mathcal{A}_c^*$.

The functions Φ_c synchronize the activities of the components. In the state reached by following w from the initial state, $\Phi_c(w, \alpha)$ is the sequence of computations that will be followed by component c when the composite system computes α. We say M is well defined if $w \in \mathbf{L} \rightarrow \phi_i(\langle\rangle, w) \in \mathbf{L}_i$, where $\mathbf{L}_i$ is the language of M_i. In other words, the traces of the language of a well defined structure correspond to traces of the component structures. The structure we have just defined is derived from the general automaton product that is described in [Gecseg] (pp. 14-15).

We can now define a final satisfaction relation $\models''$ on a $\mathcal{L}(\mathcal{A}, C)$ propositions and semantic structures.

Definition 8: We say a pair (M, w) where M is $\mathcal{L}(\mathcal{A}, C)$ semantic structure $M = (\mathbf{L}, M_{c_1}, \Phi_{c_1}, \ldots M_{c_n}, \Phi_{c_n})$, and w is a string over $\mathcal{A}$, *satisfies* a $\mathcal{L}(\mathcal{A}, C)$ proposition p, (written $(M, w) \models'' p$), iff:

1) p is in $\mathcal{L}(\mathcal{A})$ and $(\mathbf{L}, w) \models' p$,

2) $p \equiv D(\alpha, c, y)$ and $\Phi_c(w, \alpha) = y$,

3) $p \equiv (\text{in } c)q$ and $(M_c, \Phi_c(\langle\rangle, w)) \models'' q$,

4) p is a a conjunction or negation and the usual conditions apply.

Note that $\models''$ is defined so that given $q \rightarrow p$ and $(\text{in } c)q$ we can conclude $(\text{in } c)p$.

We have composite grammars in $\mathcal{L}(\mathcal{A},\mathcal{C})$ that are analogous to the unitary grammars of $\mathcal{L}(\mathcal{A})$. Composite grammars can be considered to be finite trees, with the leaves being unitary grammars. Composite grammars are conjuncts of *E rules*, *D rules* and *type definitions*. *E rules* are similar to the rules of unitary grammars. *D rules* axiomatize the Φ functions and *type definitions* associate grammars with the components.

Definition 9: . A *D rule* is an assertion $p \leftrightarrow D(\alpha, c, y)$ where p is a proposition with no subformulae $E(\beta)$ or $(\textbf{after } x)p$.

Definition 10: . A *type definition* an assertion $(\text{in } c)p$.

Definition 11: . An *E rule* is a proposition of the form $(Guard(\alpha) \wedge p) \leftrightarrow E(\alpha)$ where p is a proposition with no subformulae $E(\beta)$ or $(\textbf{after } x)p$ and $Guard(\alpha) = (\forall c \in C)D(\alpha, c, y) \rightarrow (\text{in } c)E(y)$. The guard makes sure that structures satisfying grammars will be well defined.

Definition 12: . A *composite grammar* is a conjunction

$\wedge_{c \in C}(T_c) \wedge_{\alpha \in \mathcal{A}}(R_{\mathcal{A}}) \wedge_{c \in C, \alpha \in \mathcal{A}}(R_{c,\alpha})$

where: Each T_c is a type definition, each R_α is a *E rule*, and each $R_{c,\alpha}$ is a *D rule*.

Theorem 3: . If every type definition of grammar $\mathcal{G}$ uniquely defines a component semantic structure, then $\mathcal{G}$ uniquely defines a composite semantic structure. In particular, if each type definition is of the form $(\text{in } c)\mathcal{H}$ where $\mathcal{H}$ is a unitary grammar, then $\mathcal{G}$ defines a unique composite semantic structure.

Theorem 4: . For every prefix closed regular language **L** over $\mathcal{A}$, there is a composite grammar $\mathcal{G}$ so that **L** is the language of the semantic structure defined by $\mathcal{G}$.

Proof. Let **M** be a state machine that accepts **L**, and let S,$\mathcal{A}$ and δ be, respectively, the set of states, the alphabet and the transition function of **M**. Define a unitary grammar $\mathcal{G}_1$ with alphabet S. We can define a function:

$$State() = s \Leftrightarrow (\forall s' \neq s)P(s', 1, s, 1) \bigvee \{s = initial \bigwedge (\forall s') \neg P(initial, 1, s', 1)\}$$

which returns either the most recent computation s or the initial state of **M** if no computation has yet taken place. The rules of $\mathcal{G}_1$ are then constructed so that $E(s) \Leftrightarrow (\exists \alpha \in \mathcal{A})\delta(State(), \alpha) = s$ — if and only if state s is reachable from the current state in one transition. Define a composite grammar with alphabet $\mathcal{A}$ and a single type definition $(\text{in } c)\mathcal{G}_1$. Define a function $Cstate() = s \Leftrightarrow (\text{in } c)State() = s$. The *D rules* are constructed so that $D(\alpha, c, \langle s \rangle) \Leftrightarrow \delta(Cstate(), \alpha) = s$. The *E rules* are constructed so that $E(\alpha)$ iff $\delta(Cstate(), \alpha)$ is defined. Since all the functions we employ in this construction are finite, they can be defined in the propositional logic.

References

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

[Kripke] Kripke, S. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16, pp. 83-94, 1963.

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

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

[Yodaiken] A modal recursive arithmetic of digital systems. Technical Report, Department of Computer Science, University of Massachussets (Amherst), (forthcoming).