SRI International

THE SYNTHESIS OF DIGITAL MACHINES WITH PROVABLE EPISTEMIC PROPERTIES

Technical Note 412

April 1987

By: Stanley J. Rosenschein
Director
Leslie Pack Kaelbling
Computer Scientist

Representation and Reasoning Program
Artificial Intelligence Center
Computer and Information Sciences Division
and
Center for the Study of Language and Information

This research has been made possible by a gift from the System Development Foundation, by FMC Corporation under Contract No. 147466, and by General Motors Research Laboratories under Contract No. 50-13.

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Abstract

Researchers using epistemic logic as a formal framework for studying knowledge properties of artificial-intelligence (AI) systems often interpret the knowledge formula $K(x, \varphi)$ to mean that machine x encodes φ in its state as a syntactic formula or can derive it inferentially. If $K(x, \varphi)$ is defined instead in terms of the correlation between the state of the machine and that of its environment, the formal properties of modal system S5 can be satisfied without having to store representations of formulas as data structures. In this paper, we apply the correlational definition of knowledge to machines with composite structure and describe the semantics of knowledge representations in terms of correlation-based denotation functions. In particular, we describe how epistemic properties of synchronous digital machines can be analyzed, starting at the level of gates and delays, by modeling the machine's components as agents in a multiagent system and reasoning about the flow of information among them. We also introduce Rex, a language for computing machine descriptions recursively, and explain how it can be used to construct machines with provable informational properties.

Introduction

Many important computer applications involve the design of hardware and software that are part of a larger system embedded in a physical environment. Applications of this kind arise in process control, avionics, robotics, and artificial intelligence; in the typical case, the computer's principal task is to keep track of and react to conditions in the environment. For the system to operate as desired, it must be designed to recognize the relevant environmental conditions and to compute appropriate responses when required. As more open-ended environments are considered and as conditions to be recognized and the responses to be supplied become more complex, the job of designing real-time embedded systems becomes correspondingly more difficult.

The problem is particularly acute in the design of highly reactive artificial-intelligence (AI) systems, such as intelligent robots. A robot can be viewed abstractly as a complex control system that monitors sensory inputs and acts to achieve or maintain certain goal conditions in its environment. In simple control systems, facts about the environment can often be encoded as a small set of numerical parameters. More complex kinds of information, however, such as those needed by intelligent robots, require correspondingly more complex data structures for their encoding. Moreover, real-time performance requires that there be a constant bound on the number of computational operations performed between input and output.

The Artificial Intelligence Center at SRI International is designing and implementing a mobile robot in the tradition of Shakey [19]. The aim of this project is to combine significant perceptual, reasoning, and communicational abilities in an autonomous computer-controlled device and to have it operate in real time. Because the concept of *knowledge* is a useful and powerful abstraction for the design of complex agents, we are attempting to reconcile the goal of manipulating complex information with that of real-time operation by adopting a design approach based on the use of (1) epistemic logic in the formal analysis of the robot's information states and (2) a language for specifying real-time control programs that are amenable to this type of formal analysis.

One useful abstraction in the design of such systems is the concept of *knowledge*. The statement "system x *knows* φ" provides a compact description of the *propositional content* of information encoded in x's state without specifying the details of the encoding. Much work on formalizing properties of knowledge has been done in philosophy [8,13], theoretical computer science [6], and AI [18,15,12]. Most of the work in this tradition is carried out in an abstract setting; the essential concept of *knowledge* is not given a concrete physical or computational interpretation. When such interpretations have been given for AI systems, they have typically involved encoding sentences of a formal language as data structures in the machine. For instance, a system might be regarded as knowing φ if its knowledge base contained a sentence expressing φ, or if such a sentence could be derived computationally from other sentences in the knowledge base. One important practical advantage in this approach is the ease with which the designer can attribute propositional interpretations to the machine's states. However, there are two important disadvantages of this approach:

first, general inference is highly computationally complex and cannot be carried out in real time; second, although the designer of a system may have in mind a particular interpretation of the formal language he employs in the machine, it may not describe the machine's state of information about the world correctly.

The situated-automata approach attempts to avoid inferential complexity by providing a concrete computational model of knowledge in a framework that does not depend on viewing the system as manipulating sentences of a logic [20]. In the situated-automata framework, the concept of knowledge is analyzed in terms of logical relationships between the state of a process (e.g., a machine) and that of its surrounding world. Because of constraints between a process and its environment, not every state of the process-environment pair is possible, in general. A process π is said to know a proposition φ in a situation in which its internal state is s, if in all possible situations in which π is in state s, φ is satisfied. This definition of knowledge satisfies the axioms of modal system S5, including deductive closure and positive and negative introspection.

In its original formulation, situated-automata theory dealt with the state of a system as an unanalyzed whole. Since machines designed for real applications can take on an enormous number of states, they must be built hierarchically, with the size of the state set growing as the product of the sizes of the state sets of the component machines. This paper extends situated-automata theory to hierarchically constructed machines in order to facilitate the epistemic analysis of composite machines. In particular, we use the situated-automata model of knowledge to analyze synchronous digital machines by viewing their components as elements of a multiagent system and reasoning about the flow of information among these components.

Real-time performance has often proved difficult to achieve with traditional AI techniques. This difficulty stems, in part, from a failure to distinguish between two types of facts that are relevant to a robot's operation. The first of these can be called *dynamic* facts, as they involve moment-to-moment conditions of the environment. The second type comprises *permanent* or *static* facts, that is, those that are better regarded as part of a model of the environment in which the robot operates. The traditional AI approach to the encoding of information ("knowledge representation") is to think about *all* these facts as objects of the same sort and to encode them uniformly as symbolic data structures that are manipulated by the program. This approach is attractive because it seems to offer the possibility of reducing the problem of designing intelligent machines to the conceptually simpler task of encoding knowledge in a logical language and constructing programs that syntactically derive consequences of facts in a knowledge base.

As attractive as this strategy may be, its implementation involves serious technical difficulties that derive from the computational complexity of inference. It is well recognized that the more open-ended the environment, the more expressive is the logic needed to describe it and the less tractable is the problem of reasoning explicitly in the logic. In some applications, the moment-to-moment synchronization of the programs with conditions in the surrounding world can be conveniently ignored. In such domains (e.g., theorem proving, medical diagnosis, and geology), the time complexity of inference is not a critical problem;

thus, the implementation of intelligent information processing by means of conventional symbolic inference techniques is feasible.

However, in the mobile-robot domain, the permanent facts relevant to time-critical, low-level interpretation and decision-making are so complex that is it impossible to reason with them explicitly in real time. This point is hardly controversial; the assumption is generally made that, in applications of this sort, static knowledge must be "compiled in." The work described in this paper provides conceptual foundations upon which a formal theory of knowledge compilation might be built.

Considerations of real-time performance and semantic rigor have led to the development of Rex, a set of tools for constructing complex programs with formally definable epistemic properties. Instead of constructing a description of the target machine directly, the designer writes a program that, when run, computes a component-level logical description of the machine. This description can then effectively realized as circuitry, as code for a parallel machine, or as a program that simulates the machine on a sequential computer. Since synchronization with the environment lies at the heart of our definition of knowledge, the Rex tools have been designed to guarantee real-time interaction between the target machine and the environment. Of course, the Rex system itself need not be real-time since *it* is not intended to be coupled to the physical environment.

In the remainder of this paper we present a description of the theoretical background of this work, an introduction to Rex with some simple examples, and the application of situated-automata theoretic analysis to programs suggested by the mobile robot domain.

Theoretical Background

A useful theory of intelligent embedded systems must be capable of describing how certain parts of the physical world encode information about other parts over time and how their behavior exploits that information. We model this situation abstractly by constructing the requisite concepts from a small set of primitives: space, time, possibility, and truth. Our approach is to use a propositional language that is enriched with terms for processes and the values they can take on (i.e., states they can be in) and is closed under various temporal and epistemic operators. The semantic interpretation for this language is given in terms of times, locations, and possible worlds. Processes are modeled as spatial "trajectories" with cross-world identity — that is, they are identified with functions from world-time pairs to complex spatial locations. Knowledge is modeled in terms of the relationship between the states of a process and states of the environment. In the following sections, we shall discuss these basic concepts, then introduce an epistemic and a denotational method for analyzing the semantic content of states of machines; each type of analysis will be illustrated by a simple example.

Basic Concepts

Let a universe $U = (L, T, W)$, where the set L (*atomic locations*) is a topology suitable for modeling physical space, T (*times*) an ordered set of instants, and W (*possible worlds*) an abstract set of indices of possibility, i.e. possible histories or ways the world could be.

We identify the set of *processes* Π with their spatial trajectories — that is, the set of mappings $\pi : W \times T \rightarrow 2^L$. $\pi(w,t)$ denotes the set of locations occupied by process π in world w at time t. We will let Π_0 denote the set of atomic processes; a process π is atomic if, for every world w and time t, $\pi(w,t)$ is an atomic location. The set of processes inherits the structure of L; it is closed under pointwise union, intersection, and complementation; moreover, one process can be a subprocess of another. The null process is denoted by $[\,]$, and $[\pi_1, \ldots, \pi_n]$ denotes a process tuple that is made up of subprocesses $\pi_1, \ldots, \pi_n$. The *value domain* of a process π, written as D_π, is a distinguished set of mutually exclusive and exhaustive properties of that process. The *atomic value domain*, D_0, is defined to be $\bigcup_{\pi_i \in \Pi_0} D_{\pi_i}$, and complete domain D is the union $\bigcup_{\pi_i \in \Pi} D_{\pi_i}$ of the value domain of every process. The function $\hat{q} : \Pi \times W \times T \rightarrow D_\pi$ associates with each process, world, and time the value (or state) of the process in that world at that time in such a way that $\hat{q}(\pi, w, t) \in D_\pi$.

Epistemic Logic for Analysis of Machines

Language

We begin by defining the symbols of the language:

Symbols:	$P = \{start, p_1, p_2, \ldots\}$	(atomic formulas),
	$A = \{[\,], a_1, a_2, \ldots\}$	(process constants),
	$C = \{\langle\,\rangle, c_1, c_2, \ldots\}$	(value constants),
	$F = \{f_1, f_2, \ldots\}$	(function symbols),
	$\{\Delta_c\}_{c \in C}$	(delay-element predicates),
	$\{\Pi_f\}_{f \in F}$	(function-element predicates),
	$=$	(equality symbol)
	$[\,\vert\,], \langle\,\vert\,\rangle$	(pairing functions),
	$\wedge, \neg$	(Boolean connectives),
	$K, \Box_W, \Box_T, \bigcirc$	(modalities),
	$*, \dot{\circ}$	(term operators).

Next we give the formation rules:

Terms:

1. If e is a *process* (respectively *value*) *constant*, then e is a *process* (respectively *value*) *term.*
2. If e_1, e_2 are *process* (respectively *value*) *terms*, then so is $[e_1 \mid e_2]$ (respectively $\langle e_1 \mid e_2 \rangle$).
3. If x is a *process term*, then $*x$ is a *value term.*
4. If u is a *value term* and f is a *function symbol*, then $f(u)$ is a *value term.*
5. If e is a *term*, then so is $\circ e$.
6. Nothing else is a *term.*

Formulas:
1. If p is an *atomic formula*, then p is a *formula.*
2. If e_1, e_2 are *terms*, then $e_1 = e_2$ is a *formula.*
3. If c is a *value constant*, f is a *function symbol*, and x, y are *process terms*, then $\Delta_c(x, y)$ and $\Pi_f(x, y)$ are *formulas.*
4. If φ and ψ are *formulas*, then so are $(\varphi \wedge \psi)$, $\neg\varphi$, $\Box_W\varphi$, $\Box_T\varphi$, and $\bigcirc\varphi$.
5. If x is a *process term* and φ is a *formula*, then $K(x, \varphi)$ is a *formula.*
6. Nothing else is a *formula.*

The connectives $\vee, \rightarrow, \leftrightarrow$ are defined as follows: $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, and $\varphi \leftrightarrow \psi = \varphi \rightarrow \psi \wedge \psi \rightarrow \varphi$. A combined necessity operator can be formed from the time and world modalities: $\Box\varphi = \Box_W\Box_T\varphi$. Dual modal operators can also be defined: $\Diamond\varphi = \neg\Box\neg\varphi$, etc. We abbreviate $\langle u_1 \mid \langle u_2 \mid \cdots\langle\rangle\cdots\rangle\rangle$ as $\langle u_1, u_2, \cdots\rangle$ and $[x_1 \mid [x_2 \mid \cdots[]\cdots]]$ as $[x_1, x_2, \cdots]$. We sometimes take integers to be value constants and lowercase letters early in the alphabet to be process constants. We also omit parentheses according to the usual conventions.

Semantics

The semantics of our logic is given with respect to a model of space, time, possibility, and state that is a specialization of the basic concepts presented earlier. Possibility is modeled with the standard techniques of possible-worlds semantics. For each possible world and instant of time, the model assigns an atomic state to every atomic location. We model processes as occupying collections of atomic locations; these collections vary with time and possible worlds. To conveniently name subprocesses, we impose additional structure on space and state by closing the set of locations and the set of states under *pairing* operations. Formally stated, if A is a set, we define $pairs(A)$ to be the least set that contains A, includes the distinguished element nil_A, and is closed under pairing: $x, y \in pairs(A)$ implies $(x, y) \in pairs(A)$. Processes can then be formally identified with functions that map $(W \times T)$ into $pairs(L)$ that, for each world and time, specify what structure of atomic locations is occupied by a particular process.

Models: $\mathcal{M} = (U = (L, T, W), D_0, q, I = (I_P, I_A, I_C, I_F))$, where
1. W is a nonempty set of *possible worlds.*
2. T is a nonempty set of *time instants* isomorphic to the natural numbers.
3. L is a nonempty set of *atomic locations.*
4. D_0 is a nonempty set of *atomic states.*
5. $q : L \times W \times T \rightarrow D_0$ assigns atomic states to atomic locations at every world-time pair.
6. $I_P : P \rightarrow 2^{W \times T}$ interprets atomic formulas as sets of world-time pairs.
7. $I_A : A \rightarrow ((W \times T) \rightarrow pairs(L))$ interprets process constants as processes.
8. $I_C : C \rightarrow pairs(D_0)$ interprets value constants as value structures.
9. $I_F : F \rightarrow pairs(D_0) \rightarrow pairs(D_0)$ interprets function symbols as functions on value structures.

We extend the function q to pairs in the obvious way by defining $\hat{q} : pairs(L) \times W \times T \rightarrow pairs(D_0)$ as follows:

$$\begin{aligned} \hat{q}(nil_L, w, t) &= nil_{D_0}, \\ \hat{q}(\ell, w, t) &= q(\ell, w, t) \text{ for } \ell \in L, \\ \hat{q}((s_1, s_2), w, t) &= (\hat{q}(s_1, w, t), \hat{q}(s_2, w, t)) \end{aligned}$$

The denotation of a term e relative to a model and a world-time pair, written $[\![e]\!]_t^{\mathcal{M},w}$, is defined as follows (reference to the model is suppressed):

1. $[\![a]\!]_t^w = I_A(a)(w,t)$ if a is a *process constant.*
2. $[\![c]\!]_t^w = I_C(c)$ if c is a *value constant.*
3. $[\![[x \mid y]]\!]_t^w = ([\![x]\!]_t^w, [\![y]\!]_t^w)$, for $[x \mid y]$ a process pair.
4. $[\![\langle u \mid v \rangle]\!]_t^w = ([\![u]\!]_t^w, [\![v]\!]_t^w)$, for $\langle x \mid y \rangle$ a value pair.
5. $[\![*x]\!]_t^w = \hat{q}([\![x]\!]_t^w, w, t)$.
6. $[\![f(u)]\!]_t^w = I_F(f)([\![u]\!]_t^w)$.
7. $[\![\circ e]\!]_t^w = [\![e]\!]_{t+1}^w$.

Satisfaction of formulas is defined relative to a model $\mathcal{M}$ and a world-time pair w, t (again we suppress reference to the model):

1. $w,t \models p$ if $\langle w,t \rangle \in I_P(p)$, for $p \in P$.
2. $w,t \models e_1 = e_2$ if $[\![e_1]\!]_t^w = [\![e_2]\!]_t^w$.
3. $w,t \models \Delta_c(x,y)$ if for all $w' \in W, t' \in T$, $\hat{q}([\![y]\!]_0^{w'}, w', 0) = I_C(c)$ and $\hat{q}([\![y]\!]_{t'+1}^{w'}, w', t'+1) = \hat{q}([\![x]\!]_{t'}^{w'}, w', t')$.
4. $w,t \models \Pi_f(x,y)$ if for all $w' \in W, t' \in T$, $\hat{q}([\![y]\!]_{t'}^{w'}, w', t') = I_F(f)(\hat{q}([\![x]\!]_{t'}^{w'}, w', t'))$.
5. $w,t \models (\varphi \wedge \psi)$ if $w,t \models \varphi$ and $w,t \models \psi$.
6. $w,t \models \neg\varphi$ if $w,t \not\models \varphi$.
7. $w,t \models \Box_W \varphi$ if $w',t \models \varphi$ for all $w' \in W$.
8. $w,t \models \Box_T \varphi$ if $w,t' \models \varphi$ for all $t' \in T$.
9. $w,t \models \bigcirc\varphi$ if $w,t+1 \models \varphi$.
10. $w,t \models K(x,\varphi)$ iff $w',t' \models \varphi$ for all $w' \in W, t' \in T$ such that $\hat{q}([\![x]\!]_{t'}^{w'}, w', t') = \hat{q}([\![x]\!]_t^w, w, t)$.

This definition of satisfaction can be interpreted informally as follows: clauses 1 through 4 specify the interpretation of atomic propositions, with clauses 3 and 4, respectively, covering formulas that describe delay elements and functional components of machines; clauses 5 and 6 define the interpretation of the Boolean connectives in the standard fashion; a standard treatment of possibility and linear-time temporal logic is specified in clauses 7 through 9; clause 10 can be viewed as defining the semantics of the modal knowledge operator K in terms of an epistemic accessibility relation on the set of world-time pairs. A world-time pair, (w',t'), is epistemically accessible from another (w,t) if, for all the agent in (w,t) knows, he might have been in (w',t'). In logics of this type, if the accessibility relation is an

equivalence relation, the logic will satisfy the axioms of modal system S5 [9], including the axioms of deductive closure, positive introspection, and negative introspection. We have operationalized the notion of "is the same as far as the agent knows" as "is indistinguishable to the agent;" that is, the agent's state is identical in each case. We identify agents with processes, and define the accessibility relation for process π, written as $\approx_\pi$, as follows:

$$(w,t) \approx_\pi (w',t') \equiv \hat{q}(\pi,w,t) = \hat{q}(\pi,w',t')$$

Under this definition, $\approx_\pi$ is clearly an equivalence relation on $W \times T$, and the S5 axioms are satisfied [20].

A model $\mathcal{M}$ is said to simply *satisfy* a formula if and only if $\mathcal{M},w,t \models \varphi$ for all $w \in W, t \in T$. A formula is *valid* if it is satisfied by every model. A set of formulas Γ *entails* a formula φ (written $\Gamma \models \varphi$) if and only if every model that satisfies all the sentences of Γ also satisfies φ.

We do not present an axiomatic treatment of the logic; the interested reader is referred to standard treatments of modal logic [9], logics of time and knowledge [16,14], and their application to AI [18,20]. In a later section, we present informal proofs in the logical language and appeal to valid formulas and entailments involving K and other modal operators. Some of the more important ones are listed here:

1. Theorems of propositional logic and temporal logic [16,2].
2. $\models \Box_W \varphi \rightarrow \varphi$
3. $\models \Box_T \varphi \rightarrow \varphi$
4. $\models K(x,\varphi) \rightarrow \varphi$ (truth).
5. $\models K(x,\varphi \rightarrow \psi) \rightarrow (K(x,\varphi) \rightarrow K(x,\psi)))$ (consequential closure).
6. $\models K(x,\varphi) \rightarrow K(x,K(x,\varphi))$ (positive introspection).
7. $\models \neg K(x,\varphi) \rightarrow K(x,\neg K(x,\varphi))$ (negative introspection).
8. $\models *x = v \rightarrow K(x,*x = v)$ (self-awareness).
9. $\models \Delta_c(x,y) \leftrightarrow \Box(start \rightarrow *y = c \wedge * \circ y = *x)$ (delay-element axiom).
10. $\models \Pi_f(x,y) \leftrightarrow \Box(*y = f(*x))$ (functional element axiom).
11. $\varphi, \varphi \rightarrow \psi \models \psi$ (modus ponens).
12. $\varphi \models K(x,\varphi)$ (epistemic necessitation).
13. $\varphi \models \Box_W \varphi$ (alethic necessitation).
14. $\varphi \models \Box_T \varphi$ (temporal necessitation).
15. $\varphi \models \bigcirc \varphi$ (succedent necessitation).
16. $\models K(x,\varphi) \wedge K(y,\psi) \rightarrow K([x \mid y], \varphi \wedge \psi)$ (spatial monotonicity).

Modeling Machines in the Logic

Physical processes (of which the processes in the logic are idealizations) can be described in many ways. One class of descriptions specifies permanent *structural* relationships among processes and their subparts. *Behavioral* descriptions, on the other hand, specify how the states of processes vary over time. *Epistemic* descriptions comprise yet another class, specifying the information carried by processes. These descriptions are not unrelated; in

general, structural constraints entail certain behavioral properties, which in turn entail epistemic properties.

In designing computational systems, we are especially interested in *discrete* processes, i.e., processes that can be described in terms of discrete sets of locations, states, and instants of time. For example, registers in a digital computer are easily modeled as compound processes made up of flip-flop subprocesses with value domain $\{H, L\}$, with H denoting the property of being in a high-voltage state and L a low-voltage state.

A *machine* is modeled as a pair of possibly complex discrete processes subject to structural, behavioral, or epistemic constraints. The notation $m(x, y)$ means that output process y acts as a machine of type m with respect to input process x; that is, x and y satisfy the behavioral constraints imposed by m. When we wish to be concrete, we refer to these processes as *storage locations*, since they can be realized in digital hardware as physical components, such as wires and flip-flops. The physical state of a storage location x can then be modeled in the logic as the value, $*x$, of that process.

Just as complex physical machines are built up from primitive components of a few basic types, complex machine descriptions are built up from primitive constraints corresponding to the basic component types. Pure functional machines (e.g., logic gates), are modeled in the logic by function-element predicates Π_f, while delay components are modeled by delay-element predicates Δ_c. The component modeled by $\Pi_f(x, y)$ computes the primitive function $f : D_x \rightarrow D_y$ "instantaneously"; the delay component modeled by $\Delta_c(x, y)$ initially has the constant value c at its output location y; thereafter the value of y is the value that x had one time unit ago. The behavior of these components is formally characterized in formulas 9 and 10 of the previous section. Complex machines are ultimately made up of storage locations constrained to act as machines of the primitive types and may be built up through arbitrary interconnection or by means of composition operators. One complete set of such operators consists of *serial*, *parallel*, and *feedback* compositions. These have well-understood mathematical properties and have been studied extensively in the context of the theory of automata and switching circuits [7]; they will be illustrated by examples in following sections.

Epistemic Analysis of a Simple Machine

In this section we present a concrete example of applying the logic described above to a very simple machine, namely, an *and* gate. Shown in Figure 1, it has two inputs and one output. The value domain of each of the inputs and outputs is $\{0, 1\}$, and the output is the value of the function *and* applied to the inputs. The *and* function is described by

$$and(1, 1) = 1 \wedge and(1, 0) = 0 \wedge and(0, 1) = 0 \wedge and(0, 0) = 0. \tag{1}$$

The diagram can be summarized as

$$\Pi_{and}([X \mid Y], Z). \tag{2}$$

We will now prove formally that, if X knows φ when it has the value 1 and Y knows $\varphi \rightarrow \psi$ when it has the value 1, then the output, Z knows ψ when it has the value 1. Thus, although

Figure 1: An *and* gate.

the machine is only anding its values, it can be viewed as performing modus ponens at the level of propositions. Our formal assumptions about the semantics of the machine's inputs are

$$*X = 1 \quad \rightarrow \quad K(X, \varphi) \tag{3}$$
$$*Y = 1 \quad \rightarrow \quad K(Y, \varphi \rightarrow \psi) \tag{4}$$

From Theorem 9, which describes the behavior of functional machines, and the configuration of this particular machine (Formula 2 above), we can deduce

$$\Box(*Z = and(*[X \mid Y])). \tag{5}$$

From this and the definition of *and* (Formula 1) it follows that

$$* X = 1 \wedge *Y = 1 \leftrightarrow *Z = 1. \tag{6}$$

Then, invoking our assumptions about the knowledge of the inputs (Formulas 3 and 4), we have

$$* Z = 1 \rightarrow K(X, \varphi) \wedge K(Y, \varphi \rightarrow \psi). \tag{7}$$

By the epistemic axiom of truth (Theorem 4), this formula implies

$$* Z = 1 \rightarrow (\varphi \wedge \varphi \rightarrow \psi), \tag{8}$$

and hence, by standard operations of propositional logic, also implies

$$* Z = 1 \rightarrow \psi. \tag{9}$$

The following is an instance of Theorem 8, self-awareness:

$$* Z = 1 \rightarrow K(Z, *Z = 1). \tag{10}$$

Next, we apply epistemic necessitation (Theorem 12) to Formula 9, obtaining

$$K(Z, *Z = 1 \rightarrow \psi). \tag{11}$$

Using the valid propositional transformation of strengthening the antecedent of the previous formula, we have

$$*Z = 1 \rightarrow K(Z, *Z = 1 \rightarrow \psi). \tag{12}$$

Finally, we gather Formulas 10 and 12

$$*Z = 1 \rightarrow K(Z, *Z = 1) \wedge K(Z, *Z = 1 \rightarrow \psi), \tag{13}$$

move the conjunction inside the scope of the K (the validity of this step follows from Theorem 5, consequential closure)

$$*Z = 1 \rightarrow K(Z, *Z = 1 \wedge *Z = 1 \rightarrow \psi), \tag{14}$$

and then simplify by using modus ponens, which yields

$$*Z = 1 \rightarrow K(Z, \psi). \tag{15}$$

Thus we have shown that the output location Z knows ψ when it has the value 1.

The Semantics of Knowledge Representation

The need for formal semantics of knowledge representations is well recognized by AI researchers. Usually, formal semantics are attributed in the Tarskian, truth-definitional manner by specifying a denotation function that maps symbolic structures into their meanings. Traditionally, these denotation functions have been stipulated uniformly, in the sense that a symbol is viewed as meaning the same thing no matter where it is located in memory or what state of the world caused it to be there. Furthermore, the relation between the operation of the machine and the content of the representation is often ignored. In situated-automata theory, a finer-grained approach to denotation is adopted. Meanings are associated with values in a location-dependent fashion, and the denotation function depends crucially on the behavior of the machine.

In addition to the modal $K(X, \varphi)$ notation, we can use *denotation functions* that map the values of a process to their *propositional content.* We define the set of *propositions* Φ to be $2^{W \times T}$: Each element $\varphi \in \Phi$ is the set of world-time pairs in which that proposition holds. Φ has the structure of a Boolean algebra (of sets). The ordering $\sqsubseteq$ corresponds to entailment: $\varphi \sqsubseteq \varphi'$ means that φ is less general than (i.e., entails) φ'. The operations $\sqcap$, $\sqcup$, and $\neg$ correspond to intersection, union, and complementation of propositions. The *strongest postcondition* operator $\mathrm{S} : \Phi \rightarrow \Phi$ satisfies the formula: $\mathrm{S}(\varphi)(w, t+1) \equiv \varphi(w, t)$. This is the strongest proposition that must hold one time instant in the future, given that φ holds now.

For a particular process π and value $v \in D_\pi$, we define the denotation of v for π as the strongest proposition consistent with π's having value v. This proposition corresponds to the information that the process has about its environment when its value is v. We define the *denotation* function from values to propositions $\mu_\pi : D_\pi \rightarrow \Phi$ formally as

$$\mu_\pi(v) = \{(w,t) \mid \hat{q}(\pi, w, t) = v\}.$$

Denotations and knowledge are directly related in the following manner:

$$w, t \models K(\pi, \varphi) \text{ iff } \mu_\pi(\hat{q}(\pi, w, t)) \sqsubseteq \varphi$$

In the next section we will explore the relationship between denotation and machine structure, as we did with knowledge.

Machines as Inducers of Semantic Transformations

A machine can be viewed as performing a transduction from the time series of values at its input location to values at its output location. Correspondingly, at the denotational level, each machine type has associated with it a higher-order function on denotation functions, transducing the time series of propositions known at the input location to propositions known at the output location. We call this function the *semantic-transformation function* of the machine; it takes the denotation function of the input onto the denotation function of the output. We will notate the semantic-transformation function associated with machine type m by $\tau(m)$. Formally expressed,

$$m(x, y) \Rightarrow \mu_y \sqsubseteq \tau(m)(\mu_x).$$

The semantic-transformation function for any machine is determined entirely by the semantic-transformation functions of its primitive machines and by their interconnections. For the pure functional machines Π_f, the semantic-transformation function is defined in the following way:

$$\tau(\Pi_f)(\mu)(v) = \bigsqcup_{u \in f^{-1}(v)} \mu(u)$$

Essentially, the denotation function of a particular value of the output location of a functional machine is a disjunction over the denotations of all of the possible values of the input location that could have given rise to that value in the output location.

For Δ_c, the family of delay machines parameterized by c, the semantic transformation function is defined as follows:

$$\tau(\Delta_c)(\mu)(v) = \begin{cases} \varphi_0 \sqcup \mathrm{S}(\mu(v)) & \text{if } v = c \\ \mathrm{S}(\mu(v)) & \text{otherwise} \end{cases}$$

The proposition φ_0 is taken to be the strongest proposition guaranteed to be true when the machine is started; that is, $\varphi_0 = \{(w, 0) \mid w \in W\}$. The denotation of a value v at the output location is either the strongest postcondition of the denotation of v at the input location if $v \neq c$ or, if $v = c$, the disjunction of that proposition with φ_0.

The denotation function of a complex storage location $[X_1, \ldots, X_n]$ is the intersection of the denotation functions of its sublocations:

$$\mu_{[X_1,\ldots,X_n]}([u_1, \ldots, u_n]) = \bigsqcap_{1 \leq i \leq n} \mu_{X_i}(u_i).$$

It follows that information is spatially monotonic; if X is a subprocess of Y and X carries the information that φ, then so does Y. Of course the converse is in general not true, and much of the "inference" that takes place in an intelligent machine might be viewed as *information localization*, i.e., causing information carried by a large collection of storage locations to be carried by a smaller one.

In addition, all semantic-transformation functions induced by machines are monotonic. This can be seen by observing that no negations occur in the definition of any of the semantic-transformation functions; intersection and union are both monotonic functions on the domain of denotation functions. Even an inverter (i.e. a primitive Π_{not}, where $not(\mathbf{0}) = 1, not(1) = \mathbf{0}$), induces a monotonic semantic transformation function:

$$\mu_1 \sqsubseteq \mu_2 \;\Rightarrow\; \tau(\Pi_{not})(\mu_1) \sqsubseteq \tau(\Pi_{not})(\mu_2).$$

Nor is it generally the case that $\tau(\Pi_{not})(\mu)(\mathbf{0}) = \neg(\mu)(\mathbf{0})$; instead, $\tau(\Pi_{not})(\mu)(\mathbf{0}) = \mu(1)$, a different proposition entirely.

An Example of the Use of Denotation Functions to Describe Machines

In this section we will illustrate the use of denotation functions by redescribing the and-gate machine shown in Figure 1 in terms of the informational semantics of its inputs and outputs. We will suppose that, when X has the value 1, it carries that information that φ and that, when Y has the value 1, it carries the information that $\varphi \rightarrow \psi$; then we will show that, when Z has the value 1, it carries the information that ψ. Note that the denotations of any of these values may be stronger than we have indicated without its affecting the correctness of the proof. Formally stated, we are assuming that

$$\Pi_{and}([X \mid Y], Z), \tag{16}$$

$$\mu_X(1) \sqsubseteq \varphi, \tag{17}$$

and

$$\mu_Y(1) \sqsubseteq \varphi \rightarrow \psi. \tag{18}$$

Using the semantic-transformation function of Π_{and}, we can compute the denotation function of Z:

$$\mu_Z(v) = \tau(\Pi_{and})(\mu_{[X|Y]})(v) \tag{19}$$

$$= \bigsqcup_{u \in and^{-1}(v)} \mu_{[X|Y]}(u) \tag{20}$$

For $v = 1$, the only value in the inverse image of *and* is $< 1,1 >$, so

$$\begin{aligned} \mu_Z(1) &= \mu_{[X|Y]}(< 1,1 >) && (21)\\ &= \mu_X(1) \sqcap \mu_Y(1) && (22)\\ &\sqsubseteq \varphi \wedge \varphi \rightarrow \psi && (23)\\ &\sqsubseteq \psi && (24) \end{aligned}$$

Thus, when its value is 1, Z carries at least the information that ψ holds. Note that it also carries the information that φ and $\varphi \rightarrow \psi$ holds.

Semantic-Transformation Functions for Compositions of Machines

Given machines m_1 and m_2 and their corresponding semantic-transformation functions, it is possible to calculate the semantic-transformation functions of the compositions of these machines. Let $m_1 \circ m_2$ and $m_1 \parallel m_2$, respectively, denote the *serial* and *parallel* compositions of m_1 and m_2, and let $\odot m$ denote the *feedback* operator applied to m. These forms of composition are illustrated in Figure 2, and can be described behaviorally as follows:

$$\begin{aligned} (m_1 \circ m_2)(x,y) &\equiv \exists z.m_1(x,z) \wedge m_2(z,y)\\ (m_1 \parallel m_2)(x,y) &\equiv \exists z_1, z_2.y = [z_1 \mid z_2] \wedge m_1(x,z_1) \wedge m_2(x,z_2)\\ (\odot m)(x,y) &\equiv m([x \mid y], y). \end{aligned}$$

The semantic-transformation function of the serial composition of m_1 and m_2 is simply the function composition of the semantic transformation functions of m_1 and m_2; that is,

$$\tau(m_1 \circ m_2)(\mu)(v) = \tau(m_2)(\tau(m_1)(\mu))(v).$$

In the parallel case, the semantic-transformation function of the composition of m_1 and m_2 satisfies the following equation:

$$\tau(m_1 \parallel m_2)(\mu)(v) = \tau(m_1)(\mu)(v) \sqcap \tau(m_2)(\mu)(v)$$

The feedback case involves a fixpoint; $\tau(\odot m)(\mu)$ satisfies the following:

$$\tau(\odot m)(\mu)(v) = \tau(m)(\mu')(v),$$

where $\mu'([u_1,u_2]) = \mu(u_1) \sqcap \tau(\odot m)(\mu)(u_2)$.

The Semantics of Complex Representation Structures

Although the concepts presented in this paper are developed primarily at the level of assigning informational semantics to simple registers and record structures, they can be applied to the analysis of more complex data structures. Included among such data structures are those that encode expressions in knowledge representation languages, like the ones employed in many AI systems. Designers of such systems frequently conceive of the state of the

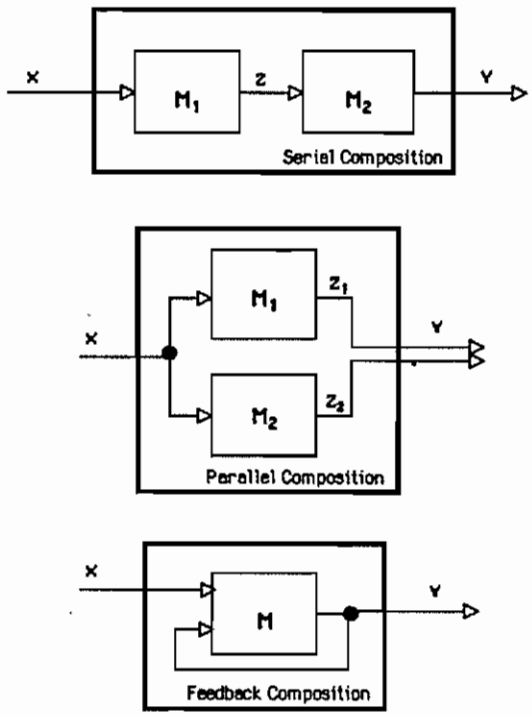


Figure 2: Serial, parallel, and feedback compositions

machine in terms of symbolic data structures that are interpreted semantically by the designer and manipulated formally by the program in a manner consistent with the designer's interpretation. One much-studied instance of this approach is the use of the language of first-order logic as a representation language and of first-order deduction as a processing strategy.

Since the theoretical concepts presented in the preceding section can be used to analyze the semantic properties of *arbitrary* machines, it follows *a fortiori* that they can be used to analyze machines executing programs that manipulate interpreted symbolic expressions drawn from some representation language. The theory determines how the denotations of various locations vary as a function of (1) the denotations of other locations upon which these locations depend and (2) the nature of the dependency (which is determined by the structure of the machine.) Let us recall that the denotation function maps values at a possible complex location into the indexical propositions, that is, propositions whose truth values depend on indices of time and possible world, that must be true when that location takes on that value.

In practice, when these relationships become too complex, it becomes awkward to specify the denotation function directly. Instead, it may be more convenient to specify the denotation function μ indirectly by positing an auxiliary domain A and expressing μ as the composition of two functions d and e, where the first function $d : D_X \rightarrow A$ maps values of a location into the auxiliary domain, while the other function, $e : A \rightarrow \Phi$, maps the auxiliary domain into propositions. The composite function is defined as $\mu(v) = e(d(v))$.

This technique of using auxiliary denotation functions is most easily illustrated by considering structured data domains, e.g. record structures. Let $Y = [P, N]$ be a compound process, where the value domain of P is $D_P = \{\mathbf{man}, \mathbf{boy}, \mathbf{woman}, \mathbf{girl}\}$ and the value domain of N is $D_N = \{0, 1, 2, \ldots\}$. (The symbols are boldfaced to emphasize that they are to be regarded as simple data values.) Rather than define the mapping from $[P, N]$ to propositions in one piece, or even define it by using the conjunctive rule in terms of independent propositional denotations of P and N, we choose two convenient nonpropositional auxiliary domains. Let auxiliary domain A_1 be some set of properties of individuals, and let A_2 be the set of natural numbers. If $d_1(\mathbf{man}) = \mathit{man}$, etc., and $d_2(\mathbf{n}) = n$, then we can specify the μ function by defining

$$e([p, n]) = \{(w, t) \mid \exists a.\ p(a)(w, t) \wedge \mathit{age}(a, w, t) = n\}$$

and setting

$$\mu_Y([u, v]) = e([d_1(u), d_2(v)]).$$

This definition implies, among other things, that

$$\mu_Y([\mathbf{girl}, 7])(w, t) \equiv \exists a.\ \mathit{girl}(a)(w, t) \wedge \mathit{age}(a, w, t) = 7.$$

This style of definition is easier for humans to understand, since intermediate-level objects can be thought of as belonging to any convenient semantic category and only at certain meaningful levels of structure aggregation are actual propositional objects considered— even though, in principle, propositional interpretations could have been attached at any level.

the same value as the other; behaviorally equivalent storage locations may be conveniently realized as the same physical storage location. Constrained storage terms are composed of a distinguished storage term and a wff that specifies the behavior of the location named by that storage term with respect to the values of other storage locations.

We will specify the semantics of the Rex forms by defining them in Lisp. Before we do this, however, we must describe some Lisp functions for manipulating the special Rex types discussed above.

(`make-cst` *st c*) This function, taking *st*, a storage term, and *c*, a constraint, returns the constrained storage term composed of *st* and *c*.

(`storage-term` *cst*) This is a simple selector function, returning the storage term associated with the constrained storage term *cst*.

(`constraint` *cst*) This selector returns the constraint associated with constrained storage term *cst*.

(`make-delta` *init next result*) This function creates a wff that specifies the initial value of the storage location denoted by *result* to be *init*, and the rest of its values to be those of the storage location denoted by *next*, delayed by one time unit. The variable *init* must represent a value of the type that may be contained in an atomic storage location (in all of our examples we will use integers as the basic value type); *next* and *result* are both storage terms. This function returns the list (`DELTA` *init next result*), which expresses the same constraint as the logical formula $\Delta_{init}(next, result)$.

(`make-pi` *fcn* (*arg*$_1$ ... *arg*$_n$) *result*) This function creates a wff that specifies that the contents of the storage location denoted by *result* be the result of applying *fcn* to the contents of the storage locations of *arg*$_1$ ... *arg*$_n$. The returned value is (`PI` *fcn* (*arg*$_1$ ... *arg*$_n$) *result*), which is equivalent to $\Pi_{fcn}([arg_1, \ldots, arg_n], result)$ in the logic.

(`make-equiv` *st*$_1$ *st*$_2$) This function creates a wff that requires that the storage locations referred to by *st*$_1$ and *st*$_2$ be behaviorally equivalent. The returned value is (`==` *st*$_1$ *st*$_2$), which is expressed in the logic by $\Box(*st_1 = *st_2)$.

(`null-stg`) This function returns the null storage term.

(`make-stg-pair` *st*$_1$ *st*$_2$) This function performs the pairing operation on storage locations. It returns a storage term that is the pair of storage terms *st*$_1$ and *st*$_2$.

(`conjoin` $c_1 \ldots c_n$) This function takes an arbitrary number of constraints and returns their conjunction.

In addition, we assume the existence of a set of standard Lisp functions, including `gensym`, which returns a new, distinct atom each time it is called.

The following table has the Rex forms in the left column, with the corresponding Lisp definitions in the right column.

(storage *name*)	(make-cst *name* ())
(plusm [cst_1 cst_2] *result*)	(conjoin (mmake-pi 'plus (list (storage-term cst_1) (storage-term cst_2)) (storage-term *result*)) (constraint cst_1) (constraint cst_2) (constraint *result*))
(init-next *init next result*)	(conjoin (make-delta *init* (storage-term *next*) (storage-term *result*)) (constraint *next*) (constraint *result*))
□	(null-stg)
[cst_1 | cst_2]	(make-cst (make-stg-pair (storage-term cst_1) (storage-term cst_2)) (conjoin (constraint cst_1) (constraint cst_2)))
(== cst_1 cst_2)	(conjoin (make-equiv (storage-term cst_1) (storage-term cst_2)) (constraint cst_1) (constraint cst_2))
(some ($v_1 \ldots v_n$) $c_1 \ldots c_m$)	((lambda (v_1 ... v_n) (conjoin c_1 ... c_m)) (gensym) ... (gensym))

In the definitions above, plusm is just one example of a number of standard arithmetic and logical primitives available to the programmer. Primitive functional machines follow the convention of being given the name of the function they compute but with a suffixed 'm.'

It is important to note that the Rex primitive forms define the way values are to be computed when the machine being specified is ultimately run. It is also necessary, however, to be able to control the specification of machines dynamically at compile time. We use the Lisp form if to create machine specifications that are conditioned on the values of Lisp expressions at compile time. Arguments that are not constraints or constrained storage terms, referred to as *value parameters*, can be used in the condition part of if forms. Note that this form is different from the ifm form, which describes a primitive functional machine with three inputs that performs a conditional computation at *run time*. The output of an ifm machine is the value of the second input if the value of the first input is 1, otherwise it is the value of the third input.

Complex functions returning constraints or constrained storage terms may be built up out of the Rex forms and defined by means of the Lisp defun form. Once such functions have been defined, a low-level machine description is calculated in two steps. The first

Notice that, under the situated-automata approach, the semantics of the representation is still regarded as being derivable from truths about the embedding of the machine in the world and thus cannot be set by the designer at will in ways that ignore those connections. For instance, in the previous example, if men are constrained to be over twenty-one years of age and if m is a machine whose semantic transform is μ_Y, it is a *theorem* (not merely a happy coincidence) that Y never takes on the value [man, 7]! Of course, in practical applications a designer may begin with an *intended* denotation function and work backwards to synthesize a particular machine that guarantees agreement between the actual semantics and the intended semantics. In the situated-automata framework, it makes no methodological sense to ignore the actual connections to the world and, consequently, to use the designer-stipulated denotations in place of the veridical information-based semantics.

Rex : A Framework for Hierarchical Machine Specification

Every machine, no matter what language it is specified in, is amenable to epistemic and denotational analysis. However, the ease of this analysis depends on the language in which the program is expressed. In this section we introduce the Rex language, which was designed to facilitate epistemic and denotational analysis. Rex programs are easier to analyze because their structure reflects the structure of the logical constraints on their behavior.

Rex is a language for the hierarchical specification of complex machines composed of primitive delay and functional elements as discussed above. A low-level machine description is computed from the Rex specification of a machine. This machine description, which stipulates how the value of each atomic storage location is to be computed over time, may then be instantiated in a variety of media (such as software and physical circuitry), making an actual machine that satisfies the initial specification.

Description of Rex

The Rex language is most easily seen as an extension of the Lisp language [17] to include forms that calculate the low-level description of a machine incrementally. This section will provide a formal description of a simple subset of Rex, which, although it has the full power of the language, is somewhat more tedious to program in than the complete version. A more practical introduction to the latter is given in a separate reference manual [11].

Rex extends Lisp by adding of a number of forms that compute machine descriptions. There are two types of Lisp values that we will use to represent these machine descriptions. They are *constraints* and *constrained storage terms*. A constraint represents a conjunction of wffs (well-formed formulas) that describe the behavior of storage locations with respect to one another. Within a constraint, storage locations are named by *storage terms*, which are typically represented by Lisp atoms. A wff may specify the way in which the value of one location is to be computed from the values of other locations or, alternatively, they may require that a pair of possibly complex storage locations be *behaviorally equivalent*. Two storage locations are behaviorally equivalent if, at every point in time, each contains

step consists of evaluating a Rex form or function that returns a constraint. Although this constraint is a machine description, it is not yet in conveniently usable form, because there are typically a large number of equivalence wffs, making it difficult to ascertain which storage terms are distinct from one another. The function makem takes a constraint as input and *canonicalizes* it, returning a useful low-level machine description.

The wffs in a noncanonical constraint can be separated into two types: equivalence constraints and pi or delta constraints. From the equivalence constraints it is possible to compute equivalence classes of storage terms by using the unification algorithm. A canonical member can then be chosen from each equivalence class; this is said to be the canonical name of the storage location represented by that equivalence class. Canonicalization is the process of computing the equivalence relation on storage terms, and substituting canonical for noncanonical names of storage locations into the rest of the wffs of the constraint. This process is illustrated by an example in the next section.

A Simple Example of Rex

In this section we present a simple Rex program and illustrate the process of computing a low-level machine description from the high-level specification. The example will be a machine with one input and one output. The input can range over integers; the output will always be zero or one. The output will be one if the machine has ever had one, two, or three as its input. The function definitions that make up the specification of the machine are shown in Figure 3.

The constant function returns a wff term that requires the storage location of const to always contain the value value. The memberm function recursively lays out circuitry to compute the membership of an element in a fixed-length list at run time; length is an integer specifying the length of list, which is a list of constrained storage terms, and item is a simple constrained storage term. The storage location of the result is constrained always to contain a 1 if the contents of item are equal to the contents of one of the elements of list, otherwise it will contain 0. The if is used at compile time to control the layout of the run time computation structure. Note also the use of == to decompose list structurally in the case that we know that it is at least one element long. The everm function returns a constraint that requires the storage location ever? to contain 1 if the contents of input have ever been 1, else to contain 0.

Our top-level function ever-a-one-two-three? returns a constraint relating the behavior of the storage locations of input and output. If the contents of the storage location of input have ever been a member of the list whose elements are the constants 1, 2, and 3, the contents of the storage location of output will be 1, else 0.

The first step in creating a low-level description of a machine satisfying this specification is to evaluate the form

```
(ever-a-one-two-three (storage 'input) (storage 'output)),
```

which will calculate a noncanonical constraint. The left column of Figure 4 contains a listing of part of the noncanonical constraint. The symbols with numeric suffixes were generated

```
(defun ever-a-one-two-three? (input output)
  (some (member? const1 const2 const3)
    (constant 1 const1)
    (constant 2 const2)
    (constant 3 const3)
    (memberm 3 [const1 const2 const3] input member?)
    (everm member? output)))

(defun everm (input ever?)
  (some (this-time-or-last)
    (orm [input ever?] this-time-or-last)
    (init-next 0 this-time-or-last ever?)))

(defun memberm (length list item member?)
  (if (= length 0)
      (constant 0 member?)
      (some (head tail equal-head? member-tail?)
        (== list [head | tail])
        (equalm [item head] equal-head?)
        (memberm (- length 1) tail item member-tail?)
        (orm [equal-head? member-tail?] member?))))

(defun constant (value const)
  (init-next value const const))
```

Figure 3: Rex code for the ever-a-one-two-three machine.

```
((PI EQUAL (ITEM1023 HEAD1027) EQUAL1029)
 (PI OR (EQUAL1029 RESULT10301031) OR1048)
 (PI EQUAL (ITEM1012 HEAD1016) EQUAL1018)
 (PI OR (EQUAL1018 RESULT10191020) OR1049)
 (PI EQUAL (ITEM1001 HEAD1005) EQUAL1007)
 (PI OR (EQUAL1007 RESULT10081009) OR1050)
 (PI OR (INPUT1053 RESULT1055) OR1056)
 (DELTA 1 DELAY976 UPDATE973)
 (DELTA 2 DELAY986 UPDATE983)
 (DELTA 3 DELAY996 UPDATE993)
 (DELTA 0 DELAY1047 UPDATE1044)
 (DELTA 0 UPDATE1059 DELAY1062)
 (== INPUT INPUT966)
 (== OUTPUT OUTPUT965)
 (== (RESULT387967 RESULT387977 RESULT387987)
     LIST1000)
 (== LIST1000 (HEAD1005 . TAIL1006))
 (== TAIL1006 LIST1011)
 (== LIST1011 (HEAD1016 . TAIL1017))
 ... )
```

```
((PI EQUAL (INPUT HEAD1027) EQUAL1029)
 (PI OR (EQUAL1029 DELAY1047) OR1048)
 (PI EQUAL (INPUT HEAD1016) EQUAL1018)
 (PI OR (EQUAL1018 OR1048) OR1049)
 (PI EQUAL (INPUT HEAD1005) EQUAL1007)
 (PI OR (EQUAL1007 OR1049) OR1050)
 (PI OR (OR1050 OUTPUT) UPDATE1059)
 (DELTA 1 HEAD1005 HEAD1005)
 (DELTA 2 HEAD1016 HEAD1016)
 (DELTA 3 HEAD1027 HEAD1027)
 (DELTA 0 DELAY1047 DELAY1047)
 (DELTA 0 UPDATE1059 OUTPUT))
```

Figure 4: Constraints resulting from definitions of `ever-a-one-two-three` machine. The left column contains a partial list of the raw constraints, the right column, the complete set of canonicalized constraints.

by gensym. Since there are 76 equational wffs, we exhibit only a few of them. The right column contains the entire constraint after it has been canonicalized.
This can be interpreted as a linear description of the wiring diagram of the "circuit" diagrammed in Figure 5.

Machine Compositions in Rex

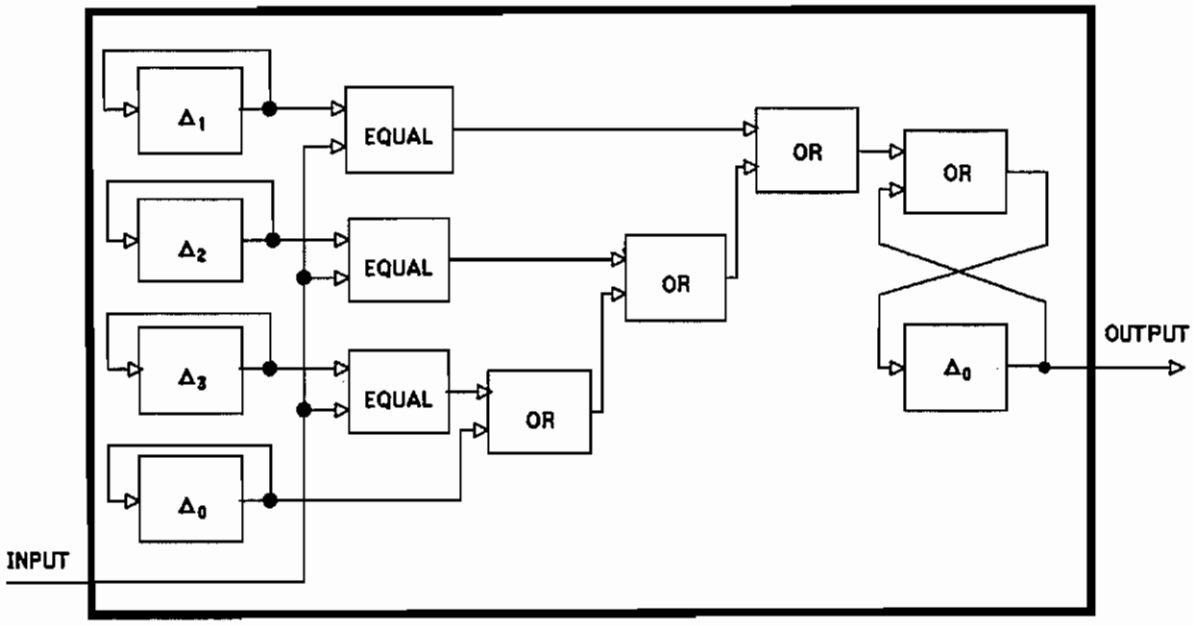
Various forms of machine composition, such as serial composition, parallel composition, and feedback, are expressed naturally in Rex. (See Figure 2 for the schematics.) Serial composition corresponds to simple relational composition:

```
(defun f (x z)
  (some (y)
    (h x y)
    (g y z)))
```

Parallel composition is achieved through pairing:

```
(defun f (x z)
  (some (y1 y2)
    (g x y1)
    (h x y2)
    (== z [y1 | y2])))
```

Feedback is achieved through cyclically dependent variables:



. Figure 5: Wiring Diagram of `ever-a-one-two-three` machine

```
(defun f (init x z)
  (some (y)
    (init-next init y z)
    (g [x | z] y)))
```

By using higher-order definitions in which machine constructors are parameters of other machine constructors, these compositions can be defined generically, though they are not often used in this form. The definitions are as follows:[1]

```
(defun serial (m1 m2 x z)
  (some (y)
    (m1 x y)
    (m2 y z)))

(defun parallel (m1 m2 x z)
  (some (y1 y2)
    (m1 x y1)
    (m2 x y2)
    (== z [y1 | y2])))

(defun f (m init x z)
  (some (y)
    (init-next init y z)
    (m [x | z] y)))
```

Epistemic and Denotational Analysis of Machines

We can carry out the analysis of a machine in terms of either denotations and the μ operator, or knowledge and the K operator. In this section we will present an example of each type of analysis.

Example of Epistemic Analysis

In this section we present the Rex specification of a robot that intermittently senses the location of a moving object in its 2-dimensional environment and tries to keep track of whether the object is within *shouting distance* (a certain radius) of the robot. Because information becomes degraded over time and by virtue of the object's motion, the machine will sometimes *know* that the object is within shouting distance, sometimes *know* that the object is *not* within shouting distance, and sometimes not know either proposition. We characterize the epistemic properties of this machine's outputs in terms of the epistemic properties of its inputs and we outline a proof of this characterization.

[1] If the implementation is in a version of Lisp that requires `funcall`, these definitions must be modified slightly to include the call explicitly.

Description of the shoutm Machine

The machine constructor has three inputs, which we will refer to as x, y, and action. The values of the storage locations denoted by x and y encode the Cartesian coordinates of the object detected by the sensor in the robot's current frame of reference; if no object is sensed, the values of the storage locations denoted by x and y are equal to the value **bottom**. The values of the storage location denoted by action encode the robot's last action; it can take on four possible values: **move** means that the robot has moved one unit in the direction it was facing; **rturn** means that the robot has turned 90 degrees to the right; **lturn** means that the robot has turned 90 degrees to the left; **noop** means that the robot has done nothing at all.

The machine constructor has two other arguments, K-shout-dist and K-not-shout-dist that denote output lines of the resulting machine. We will show that the storage location denoted by K-shout-dist *knows* that the object is within shouting distance whenever its value is 1 and that the storage location denoted by K-not-shout-dist knows that the object is *not* within shouting distance whenever its value is 1.

The top-level Rex function specifies the relation between the storage locations denoted by x, y, and action, on the one hand, and by K-shout-dist and K-not-shout-dist, on the other, by introducing complex intermediate locations, denoted by queue, K-shout-dist-vec and K-not-shout-dist-vec, and constraining them with respect to the top-level inputs and outputs. The queue will contain the most recent several sightings of the object; the number of such sightings is specified by the value parameter size. Each sighting carries some information about the current location of the object. In general, the older the sighting, the weaker the information, as the object may have moved since it was sighted (it can move one unit of distance per unit time). The queue of sightings is mapped into the storage location denoted by K-shout-dist-vec, a vector of Boolean values, the ith of which has the value 1 if the information in the ith element of the queue entails that the object is within shouting distance. Similarly, the queue is mapped into the storage location denoted by K-not-shout-dist-vec, a vector with elements that signify that the object is not within shouting distance when they carry the value 1.

```
(defun shoutm (size x y action K-shout-dist K-not-shout-dist)
  (some (queue K-shout-dist-vec K-not-shout-dist-vec)
    (queue-with-transform size [x y] action queue)
    (mapfn 'K-shoutable 0 size queue K-shout-dist-vec)
    (mapfn 'K-not-shoutable 0 size queue K-not-shout-dist-vec)
    (morm size K-shout-dist-vec K-shout-dist)
    (morm size K-not-shout-dist-vec K-not-shout-dist)))
```

To illustrate the flexibility of Rex, we parameterize the specification of the vectors K-shout-dist-vec and K-not-shout-dist-vec, along with their connection to queue, not only by size, but also by the functional value parameters K-shoutable and K-not-shoutable, which are applied to queue by mapfn.

Epistemic Properties of the shoutm Machine

The formal property we wish to prove of the shoutm machine specification is

$$\Gamma, \texttt{shoutm}(n, X, Y, A, S, Ns), \mathit{InputAxioms}(X, Y, A) \models *S = 1 \rightarrow K(S, wsd),$$

where Γ is a *background theory*—that is, a collection of wffs embodying general facts about the robot world, $\texttt{shoutm}(n, X, Y, A, S, Ns)$ is the wff computed by Rex that describes the shoutm machine, and $\mathit{InputAxioms}(X, Y, A)$ are all the instances of the following schema:

$$(*[X, Y] = \langle x, y \rangle \wedge x \neq \perp \wedge y \neq \perp \rightarrow \mathit{locf}(\langle x, y \rangle, 0)) \wedge (*A \doteq a \rightarrow \mathit{doing}(a)),$$

where $x, y \in \mathrm{N}$ and $a \in \{\mathbf{move}, \mathbf{lturn}, \mathbf{rturn}, \mathbf{noop}\}$. The atomic formula *wsd* designates the proposition "the object is within shouting distance." The formula designated by $\mathit{locf}(\langle x_i, y_i \rangle, i)$ expresses the fact that i time units ago the object was at what is location x_i,y_i in the current frame of reference. The formula $\mathit{doing}(a)$ expresses the proposition that the robot is doing the action referred to by a. We assume that Γ contains, among other things, axioms that relate *locf* to *doing*.

The truth of the epistemic specification of shoutm follows directly from three lemmas that characterize the constraints imposed by queue-with-transform, mapfn, K-shoutable, and morm. Only the proof of the third lemma will be presented; the proofs of the other two lemmas are similar in structure. We also prove an auxiliary fact relating states of processes and knowledge.

Lemma 1 Γ,$\texttt{queue-with-transform}(n, [X, Y], A, [E_1, \ldots, E_n])$, $\mathit{InputAxioms}(X, Y, A) \models$

$$\bigvee_{i=1}^{n} (*E_i = \langle x_i, y_i \rangle \rightarrow K(E_i, \mathit{locf}(\langle x_i, y_i \rangle, i)))$$

This result relates the values $\langle x_i, y_i \rangle$ of each element E_i of the queue to $\mathit{locf}(\langle x_i, y_i \rangle, i)$, taking into account the fact that the values $\langle x_i, y_i \rangle$ are interpreted relative to the robot's current frame of reference and are updated at each step as a function of the value of A.

Lemma 2 $\Gamma, \texttt{mapfn}(\texttt{K-shoutable}, 0, n, [E_1, \ldots, E_n], [F_1, \ldots, F_n])$,

$$\bigvee_{i=1}^{n} (*E_i = \langle x_i, y_i \rangle \rightarrow K(E_i, \mathit{locf}(\langle x_i, y_i \rangle, i))) \models \bigwedge_{i=1}^{n} (*F_i = 1 \rightarrow K(F_i, wsd))$$

The definition of mapfn is such that K-shoutable will be applied to each of the E_i, yielding F_i. This lemma asserts that, if the inputs to the K-shoutable machines always encode the uncertain location of the object, then, when any of the outputs has the value 1, it carries the information that the object is within shouting distance.

Lemma 3

$$\texttt{morm}(n, [F_1, \ldots, F_n], R), \bigwedge_{i=1}^{n} (*F_i = 1 \rightarrow K(F_i, \varphi)) \models \Box(*R = 1 \rightarrow K(R, \varphi))$$

If each input to the morm machine knows the proposition φ when it has the value 1, then the output also knows φ when it has the value 1.

Proof of Lemma 3

The following Rex definition is the specification of the morm (morm stands for "**m**ultiple **or** machine"). It builds the definition of a machine that computes the n-ary *or* function by accumulating $n-1$ binary orm constraints, where n is the value parameter `length`.

```
(defun morm (length vector result)
  (if (= length 0)
      (constant 0 result)
      (some (head tail result1)
        (== vector [head | tail])
        (morm (- length 1) tail result1)
        (orm [head result1] result))))
```

The proof of Lemma 3 requires the following fact about morm,

$$\mathtt{morm}(n, [F_1, \ldots, F_n], R) \models \Box([*F_1 = 1 \vee \ldots \vee *F_n = 1] \leftrightarrow *R = 1),$$

which can be easily proved by induction on the size of the input vector, `length`.

By propositional reasoning,

$$\bigwedge_{i=1}^{n}(*F_i = 1 \rightarrow K(F_i, \varphi)) \models \bigvee_{i=1}^{n} *F_i = 1 \rightarrow \bigvee_{i=1}^{n} K(F_i, \varphi).$$

Combining this with the previous fact about the values of morm, and substituting $*R = 1$ for $(*F_1 = 1 \vee \ldots \vee *F_n = 1)$, we derive

$$\mathtt{morm}(n, [F_1, \ldots, F_n], R), \bigwedge_{i=1}^{n}(*F_i = 1 \rightarrow K(F_i, \varphi)) \models *R = 1 \rightarrow \bigvee_{i=1}^{n} K(F_i, \varphi)$$

It follows from the epistemic axiom of truth and the properties of disjunction that $K(F_1, \varphi) \vee \ldots \vee K(F_n, \varphi) \rightarrow \varphi$, so we can derive

$$\mathtt{morm}(n, [F_1, \ldots, F_n], R), \bigwedge_{i=1}^{n}(*F_i = 1 \rightarrow K(F_i, \varphi)) \models (*R = 1 \rightarrow \varphi).$$

Since $P \models \Box P$ and $\models (\Box(*x = v \rightarrow \varphi)) \rightarrow (*x = v \rightarrow K(x, \varphi))$ (see proof below),

$$\mathtt{morm}(n, [F_1, \ldots, F_n], R), \bigwedge_{i=1}^{n}(*F_i = 1 \rightarrow K(F_i, \varphi)) \models (*R = 1 \rightarrow K(R, \varphi)).$$

All that remains is to verify the validity of

$$(\Box(*x = v \rightarrow \varphi)) \rightarrow (*x = v \rightarrow K(x, \varphi)).$$

We begin by observing the general fact about the epistemic properties of machines that all agents know all necessary truths: $\Box\varphi \rightarrow \Box K(x, \varphi)$. As a consequence of this, we can conclude

$$\Box(*x = v \rightarrow \varphi) \rightarrow K(x, *x = v \rightarrow \varphi)$$

This fact, together with the self-awareness axiom,

$$*x = v \rightarrow K(x, *x = v),$$

implies that

$$\Box(*x = v \rightarrow \varphi) \wedge *x = v \rightarrow K(x, *x = v \rightarrow \varphi) \wedge K(x, *x = v).$$

As an instance of the consequential-closure axiom-schema of epistemic logic, we have

$$K(x, *x = v \rightarrow \varphi) \wedge K(x, *x = v) \rightarrow K(x, \varphi).$$

Chaining the two preceding statements, we have

$$\Box(*x = v \rightarrow \varphi) \wedge *x = v \rightarrow K(x, \varphi),$$

which directly implies the result

$$\Box(*x = v \rightarrow \varphi) \rightarrow (*x = v \rightarrow K(x, \varphi)).$$

Informational Semantics of a Machine with Feedback

In this section we will describe a simple machine that continuously tracks the orientation of a robot, and we will prove that our intuitive attribution of those semantics to its outputs is in fact correct. We will assume the existence of a robot with the same motor capabilities as the one in the previous example: at each time instant, it can move forward one unit of distance, turn 90 degrees to the left or right, or remain stationary. The following Rex program specifies a machine that is intended to keep track of this robot's orientation.

```
(defun orientation (cmd orient)
  (some* (e1 e2 sub1 add1 lmod4 rmod4 if1 if2)
    (equalm cmd !lturn e1)
    (equalm cmd !rturn e2)
    (minusm orient !1 sub1)
    (plusm orient !1 add1)
    (modm sub1 !4 lmod4)
    (modm add1 !4 rmod4)
    (ifm e2 rmod4 orient if1)
    (ifm e1 lmod4 if1 if2)
    (init-next 0 if2 orient)))
```

We use the construction !*val* to abbreviate `(constant val)`, that is, to stand for a storage location that always contains the value *val*. The full Rex programming language includes functional expressions, such as `(plusm (timesm a b) c)`, which make definitions less tedious and more compact, but complicate the analysis somewhat. The schematic diagram of this program is shown in Figure 6; the canonical machine description for this program is

```
((PI EQUAL (LTURN CMD) E1)
 (PI EQUAL (RTURN CMD) E2)
 (PI MINUS (ORIENT D1) SUB1)
 (PI PLUS (ORIENT D1) ADD1)
 (PI MOD (SUB1 D4) LMOD4)
 (PI MOD (ADD1 D4) RMOD4)
 (PI IF (E2 RMOD4 ORIENT) IF1)
 (PI IF (E1 LMOD4 IF1) IF2)
 (DELTA 0 IF2 ORIENT)
 (DELTA 'lturn LTURN LTURN)
 (DELTA 'rturn RTURN RTURN)
 (DELTA 1 D1 D1)
 (DELTA 4 D4 D4))
```

The value domains of the machine's input, cmd, and its output, orient, are

$$\begin{aligned} D_{\texttt{cmd}} &= \{\textbf{move}, \textbf{lturn}, \textbf{rturn}, \textbf{noop}\} \\ D_{\texttt{orient}} &= \{\textbf{0}, \textbf{1}, \textbf{2}, \textbf{3}\} \end{aligned}$$

We will stipulate the denotation function of cmd to be

$$\begin{aligned} \mu_{\texttt{cmd}}(\textbf{move}) &= \textit{moving} \\ \mu_{\texttt{cmd}}(\textbf{lturn}) &= \textit{turning\_left} \\ \mu_{\texttt{cmd}}(\textbf{rturn}) &= \textit{turning\_right} \\ \mu_{\texttt{cmd}}(\textbf{noop}) &= \textit{still} \end{aligned}$$

and assume the following facts about the environment:

$$\varphi_0 \sqsubseteq \textit{facing\_north}$$

$$\begin{aligned} \mathrm{S}(\textit{facing\_north} \sqcap \textit{moving}) &\sqsubseteq \textit{facing\_north} \\ \mathrm{S}(\textit{facing\_east} \sqcap \textit{moving}) &\sqsubseteq \textit{facing\_east} \\ \mathrm{S}(\textit{facing\_south} \sqcap \textit{moving}) &\sqsubseteq \textit{facing\_south} \\ \mathrm{S}(\textit{facing\_west} \sqcap \textit{moving}) &\sqsubseteq \textit{facing\_west} \\ \mathrm{S}(\textit{facing\_north} \sqcap \textit{still}) &\sqsubseteq \textit{facing\_north} \\ \mathrm{S}(\textit{facing\_east} \sqcap \textit{still}) &\sqsubseteq \textit{facing\_east} \\ \mathrm{S}(\textit{facing\_south} \sqcap \textit{still}) &\sqsubseteq \textit{facing\_south} \\ \mathrm{S}(\textit{facing\_west} \sqcap \textit{still}) &\sqsubseteq \textit{facing\_west} \\ \mathrm{S}(\textit{facing\_north} \sqcap \textit{turning\_left}) &\sqsubseteq \textit{facing\_west} \\ \mathrm{S}(\textit{facing\_east} \sqcap \textit{turning\_left}) &\sqsubseteq \textit{facing\_north} \\ \mathrm{S}(\textit{facing\_south} \sqcap \textit{turning\_left}) &\sqsubseteq \textit{facing\_east} \end{aligned}$$

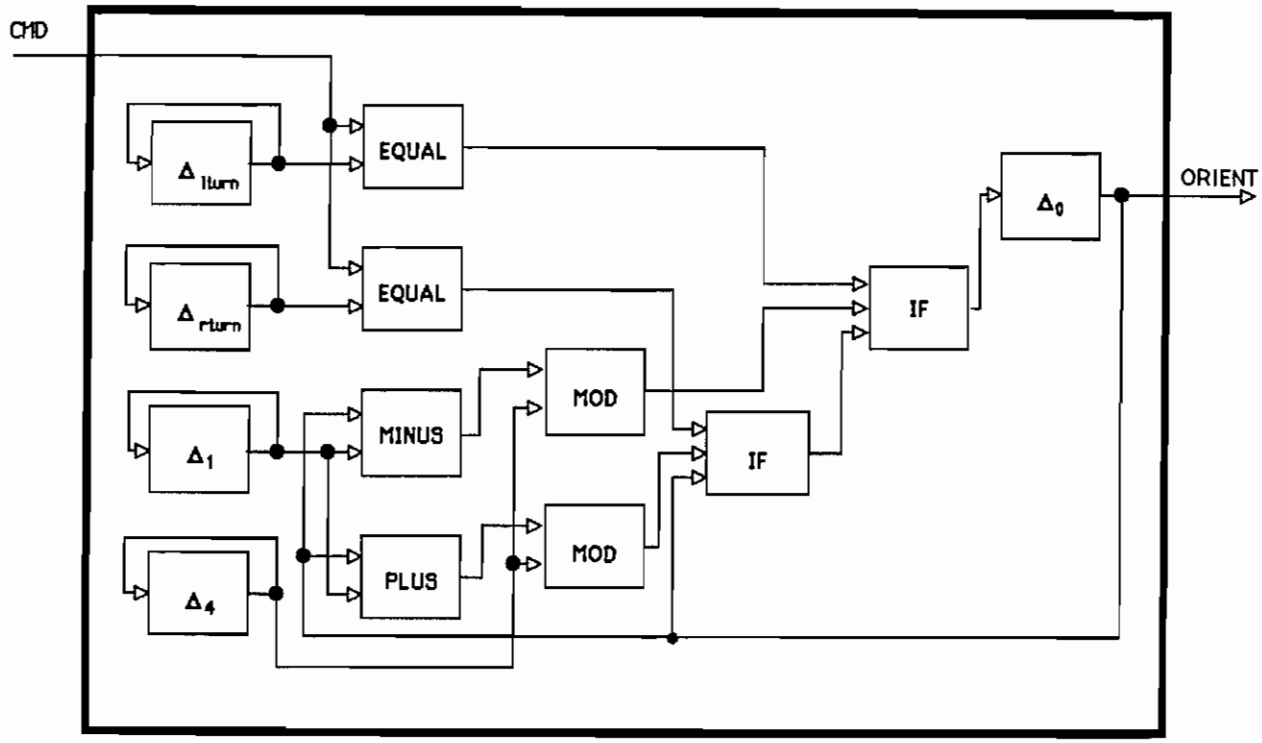


Figure 6: Wiring diagram of `orientation` machine

$$
\begin{aligned}
\mathrm{S}(\mathit{facing\_west} \sqcap \mathit{turning\_left}) &\sqsubseteq \mathit{facing\_south} \\
\mathrm{S}(\mathit{facing\_north} \sqcap \mathit{turning\_right}) &\sqsubseteq \mathit{facing\_east} \\
\mathrm{S}(\mathit{facing\_east} \sqcap \mathit{turning\_right}) &\sqsubseteq \mathit{facing\_south} \\
\mathrm{S}(\mathit{facing\_south} \sqcap \mathit{turning\_right}) &\sqsubseteq \mathit{facing\_west} \\
\mathrm{S}(\mathit{facing\_west} \sqcap \mathit{turning\_right}) &\sqsubseteq \mathit{facing\_north}
\end{aligned}
$$

We also assume that the states *moving*, *still*, *turning_left*, and *turning_right* are exhaustive and mutually exclusive—that is, for example,

$$\mathit{moving} \sqsubseteq \neg \mathit{still} \sqcap \neg \mathit{turning\_left} \sqcap \neg \mathit{turning\_right}$$

Now we will prove that the denotation function of `orient` satisfies the following definition of $\mu_{\texttt{orient}}$,

$$
\begin{aligned}
\mu_{\texttt{orient}}(\mathbf{0}) &\sqsubseteq \mathit{facing\_north} \\
\mu_{\texttt{orient}}(\mathbf{1}) &\sqsubseteq \mathit{facing\_east} \\
\mu_{\texttt{orient}}(\mathbf{2}) &\sqsubseteq \mathit{facing\_west} \\
\mu_{\texttt{orient}}(\mathbf{3}) &\sqsubseteq \mathit{facing\_south}
\end{aligned}
\quad ,
$$

by showing that it is a fixpoint of the semantic-transformation function associated with the `orientation` machine. We will proceed by assuming that `orient`, seen as an input to the machine, has these semantics, and we will calculate the semantics of its output by computing the denotation functions of each intermediate locations in the machine.

To avoid tedious details, we will assume that the locations `LTURN`, `RTURN`, `D1` and `D4` always contain the values **lturn**, **rturn**, **1**, and **4**, respectively, and that they carry no information. We may assume that they carry no information, because they are consistent with every possible state of the environment, and so do not provide the machine with any further power for discriminating the actual state of the environment. Using these assumptions, we can compute the denotation functions of `e1` and `e2`:

$$
\begin{aligned}
\mu_{\texttt{e1}}(\mathbf{0}) &\sqsubseteq \bigsqcup_{u \in \mathrm{equal}^{-1}(\mathbf{0})} \mu_{[\texttt{lturn|cmd}]}(u) \\
&\sqsubseteq \mu_{\texttt{cmd}}(\mathbf{rturn}) \sqcup \mu_{\texttt{cmd}}(\mathbf{move}) \sqcup \mu_{\texttt{cmd}}(\mathbf{noop}) \\
&\sqsubseteq \mathit{turning\_right} \sqcup \mathit{moving} \sqcup \mathit{still} \\
&\sqsubseteq \neg \mathit{turning\_left}
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
\mu_{\texttt{e1}}(\mathbf{1}) &\sqsubseteq \mathit{turning\_left} \\
\mu_{\texttt{e2}}(\mathbf{0}) &\sqsubseteq \neg \mathit{turning\_right} \\
\mu_{\texttt{e2}}(\mathbf{1}) &\sqsubseteq \mathit{turning\_right}
\end{aligned}
$$

Now we use the assumed denotation function of `orient` to compute the denotation functions of `sub1` and `add1`:

$$\begin{aligned}\mu_{\texttt{sub1}}(-1) &\sqsubseteq \bigsqcup_{u \in \texttt{minus}^{-1}(-1)} \mu_{[\texttt{orient}|\texttt{d1}]}(u) \\ &\sqsubseteq \mu_{\texttt{orient}}(0) \\ &\sqsubseteq \mathit{facing\_north}\end{aligned}$$

Similarly,

$$\begin{aligned}\mu_{\texttt{sub1}}(0) &\sqsubseteq \mathit{facing\_east} \\ \mu_{\texttt{sub1}}(1) &\sqsubseteq \mathit{facing\_south} \\ \mu_{\texttt{sub1}}(2) &\sqsubseteq \mathit{facing\_west}\end{aligned}$$

$$\begin{aligned}\mu_{\texttt{add1}}(1) &\sqsubseteq \mathit{facing\_north} \\ \mu_{\texttt{add1}}(2) &\sqsubseteq \mathit{facing\_east} \\ \mu_{\texttt{add1}}(3) &\sqsubseteq \mathit{facing\_south} \\ \mu_{\texttt{add1}}(4) &\sqsubseteq \mathit{facing\_west}\end{aligned}$$

The values of `lmod4` and `rmod4` are computed by taking the values of `sub1` and `add1` modulo 4; since no information is added by this operation, the denotation functions are changed only slightly:

$$\begin{aligned}\mu_{\texttt{lmod4}}(0) &\sqsubseteq \bigsqcup_{u \in \texttt{mod}^{-1}(0)} \mu_{[\texttt{sub1}|\texttt{d4}]}(u) \\ &\sqsubseteq \mu_{\texttt{sub1}}(0) \\ &\sqsubseteq \mathit{facing\_east}\end{aligned}$$

Similarly,

$$\begin{aligned}\mu_{\texttt{lmod4}}(1) &\sqsubseteq \mathit{facing\_south} \\ \mu_{\texttt{lmod4}}(2) &\sqsubseteq \mathit{facing\_west} \\ \mu_{\texttt{lmod4}}(3) &\sqsubseteq \mathit{facing\_north}\end{aligned}$$

$$\begin{aligned}\mu_{\texttt{rmod4}}(0) &\sqsubseteq \mathit{facing\_west} \\ \mu_{\texttt{rmod4}}(1) &\sqsubseteq \mathit{facing\_north} \\ \mu_{\texttt{rmod4}}(2) &\sqsubseteq \mathit{facing\_east} \\ \mu_{\texttt{rmod4}}(3) &\sqsubseteq \mathit{facing\_south}\end{aligned}$$

Now we compute the denotation functions for the output of the first `if` machine:

$$\begin{aligned}
\mu_{\texttt{if1}}(\mathbf{0}) &\sqsubseteq \bigsqcup_{u \in \texttt{if}^{-1}(\mathbf{0})} \mu_{\texttt{[e2,rmod4,orient]}}(u) \\
&\sqsubseteq (\mu_{\texttt{e2}}(\mathbf{1}) \sqcap \mu_{\texttt{rmod4}}(\mathbf{0})) \sqcup (\mu_{\texttt{e2}}(\mathbf{0}) \sqcap \mu_{\texttt{orient}}(\mathbf{0})) \\
&\sqsubseteq (\mathit{turning_right} \sqcap \mathit{facing_west}) \sqcup (\neg \mathit{turning_right} \sqcap \mathit{facing_north})
\end{aligned}$$

Similarly,

$$\begin{aligned}
\mu_{\texttt{if1}}(\mathbf{1}) &\sqsubseteq (\mathit{turning_right} \sqcap \mathit{facing_north}) \sqcup (\neg \mathit{turning_right} \sqcap \mathit{facing_east}) \\
\mu_{\texttt{if1}}(\mathbf{2}) &\sqsubseteq (\mathit{turning_right} \sqcap \mathit{facing_east}) \sqcup (\neg \mathit{turning_right} \sqcap \mathit{facing_south}) \\
\mu_{\texttt{if1}}(\mathbf{3}) &\sqsubseteq (\mathit{turning_right} \sqcap \mathit{facing_south}) \sqcup (\neg \mathit{turning_right} \sqcap \mathit{facing_west})
\end{aligned}$$

The denotation function for the second `if` machine depends on that of the first:

$$\begin{aligned}
\mu_{\texttt{if2}}(\mathbf{0}) &\sqsubseteq \bigsqcup_{u \in \texttt{if}^{-1}(\mathbf{0})} \mu_{\texttt{[e1,lmod4,if1]}}(u) \\
&\sqsubseteq (\mu_{\texttt{e1}}(\mathbf{1}) \sqcap \mu_{\texttt{lmod4}}(\mathbf{0})) \sqcup (\mu_{\texttt{e1}}(\mathbf{0}) \sqcap \mu_{\texttt{if1}}(\mathbf{0})) \\
&\sqsubseteq (\mathit{turning_left} \sqcap \mathit{facing_east}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \mathit{turning_right} \sqcap \mathit{facing_west}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \neg \mathit{turning_right} \sqcap \mathit{facing_north})
\end{aligned}$$

Similarly,

$$\begin{aligned}
\mu_{\texttt{if2}}(\mathbf{1}) &\sqsubseteq (\mathit{turning_left} \sqcap \mathit{facing_south}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \mathit{turning_right} \sqcap \mathit{facing_north}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \neg \mathit{turning_right} \sqcap \mathit{facing_east}) \\
\mu_{\texttt{if2}}(\mathbf{2}) &\sqsubseteq (\mathit{turning_left} \sqcap \mathit{facing_west}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \mathit{turning_right} \sqcap \mathit{facing_east}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \neg \mathit{turning_right} \sqcap \mathit{facing_south}) \\
\mu_{\texttt{if2}}(\mathbf{3}) &\sqsubseteq (\mathit{turning_left} \sqcap \mathit{facing_north}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \mathit{turning_right} \sqcap \mathit{facing_south}) \sqcup \\
&\quad (\neg \mathit{turning_left} \sqcap \neg \mathit{turning_right} \sqcap \mathit{facing_west})
\end{aligned}$$

Finally, we compute the effect of the `delta` machine on the denotation function of `if2`, making use of the assumed connections between the facings and actions of the robot:

$$\mu_{\texttt{orient}}(\mathbf{0}) \sqsubseteq \varphi_0 \sqcup \mathbf{S}(\mu_{\texttt{if2}}(\mathbf{0}))$$

$$
\begin{aligned}
&\sqsubseteq\ \mathit{facing\_north} \sqcup \\
&\quad \mathsf{S}((\mathit{turning\_left} \sqcap \mathit{facing\_east}) \sqcup \\
&\quad (\neg \mathit{turning\_left} \sqcap \mathit{turning\_right} \sqcap \mathit{facing\_west}) \sqcup \\
&\quad (\neg \mathit{turning\_left} \sqcap \neg \mathit{turning\_right} \sqcap \mathit{facing\_north})) \\
&\sqsubseteq\ \mathit{facing\_north} \sqcup \mathit{facing\_north} \sqcup \mathit{facing\_north} \sqcup \mathit{facing\_north} \\
&\sqsubseteq\ \mathit{facing\_north}
\end{aligned}
$$

To compute the function on the other values, we perform the same process as above, but without disjoining in the condition φ_0:

$$
\begin{aligned}
\mu_{\texttt{orient}}(1) &\sqsubseteq \mathit{facing\_east} \\
\mu_{\texttt{orient}}(2) &\sqsubseteq \mathit{facing\_south} \\
\mu_{\texttt{orient}}(3) &\sqsubseteq \mathit{facing\_west}
\end{aligned}
$$

Thus, we have shown that the function defined above is a fixpoint of the semantic-transformation function of the `orientation` machine and, therefore, is the denotation function of `orient`.

Related Work, Implementation, and Future Directions

Related Work

There has been much work on formalizing properties of knowledge in philosophy [8,13], theoretical computer science [6,14], and AI [18,15,12]. Halpern and Lehmann have used epistemic logics to characterize the states of knowledge of a collection of processors in a distributed computer system as a result of communication between them; although their motivations are different, their technical formulations are close to ours.

Our approach to circuit synthesis is somewhat similar to work by Johnson [10] on the synthesis of digital circuits from recursion equations. Johnson's work is based on the transformation of recursive behavioral specifications of a circuit into realizations. The epistemic and denotational analysis is new, however. Analogous methods have been employed by Hillis and Chapman for circuit design [1], and by Goad for model-based vision [3]. Rex also bears some resemblance to dataflow languages (e.g. Lucid [21]), although our semantics is location-oriented rather than stream-oriented as in Lucid and other dataflow languages. Mike Gordon has developed a higher-order-logic approach to the specification and verification of hardware [4,5] that is functionally similar to the Rex method, allowing parameterized, recursive descriptions of circuits.

Implementation of the Rex System

The Rex system has been implemented in Zetalisp and CommonLisp and is currently running on the Symbolics 3600, DEC 2060, and Sun workstation. Rex is implemented as an extension to Lisp making use of Lisp's macro facility for special syntactic forms. Rex

definitions result in the creation of Lisp functions that construct machine descriptions by collecting and propagating constraints on storage locations of the target machine. Equational constraints are resolved by using a variant of the unification algorithm. An abstract machine description computed by Rex may be realized in digital hardware, since it is virtually a circuit diagram and seems well suited for implementation on fine-grained parallel architectures, such as the Connection Machine. However, it is also suitable for realization as code in conventional languages for sequential hardware. Our current implementation, for instance, supports code generation in Lisp, C, and MC68010 machine language.

Future Directions

We see at least three major directions in which the situated-automata approach to knowledge representation can be extended and applied: compilation of knowledge, weakening of knowledge to belief, and exploration of the connections between the usual notion of "reasoning" and the localization of information within a machine.

Knowledge Compilation

Thus far, the only examples of the synthesis of machines that we have presented have been performed manually, with their semantics justified by appeals to facts in the "background theory,"—namely, to those things we assume to be true in the environment of the machine. It would be of great utility to be able to specify formally the background theory, the denotation function of the inputs, and the intended denotation function of the outputs, and to use a Rex program to compile them into a fixed machine. It should be noted that truly static processes, such as an unchanging assertional database or fixed grammar rules, carry no information beyond φ_0; hence they may be encoded directly as constraints among those processes that *do* vary over time. This would allow us to use at compile time the static structures that are typically employed in run-time symbolic processing, and thus create an efficient machine that does no superfluous symbolic processing at run time.

Weakening the Notion of Knowledge to Belief

In many practical cases, when the concept of knowledge is too strong to describe a machine's actual information, we would prefer to work with the weaker notion of *belief*. There are at least two ways of defining the concept of belief in the situated-automata theoretic framework: exception hierarchies and probabilistic "degrees of belief." These formulations have direct analogues in standard AI practice.

In the exception-hierarchy approach, belief can be defined in terms of positive and negative knowledge conditions, allowing us to build machines that "jump to conclusions" based on lack of knowledge and then RETRACT them automatically as new knowledge is gained. For example, working in a modal language, we can introduce axioms like the following for each specific φ of interest:

$$B(X,\varphi) \equiv K(X,\varphi) \vee (\neg K(X,\varphi) \wedge \neg K(X,\neg\varphi) \wedge B(X,\varphi'))$$

where φ' is a condition that provides sufficient evidence for X to believe φ. Eventually the conditions ground out in positive and negative K formulas. $B(X,\varphi)$ is clearly nonmonotonic; increased information can falsify the B condition.

In the probabilistic approach, we define the correlation between states of a machine and states of the world in statistical terms. An agent X in state v is said to believe φ to degree α (written $B_\alpha(X,\varphi)$) if the conditional probability of φ, given that X is in state v, is at least α; this is the case if φ holds in 100α percent of X's epistemically accessible possible worlds. Of course, the property $B(X,\varphi) \rightarrow \varphi$ fails because φ can have a very high conditional probability, given that state of x, and still be false.

The nonmonotonic nature of conditional probabilities is well known and one direct consequence is the failure of the spatial monotonicity principle for the B operator, since the conditional probability of a proposition φ, given the states of x and y separately, bears little relation, in general, to the conditional probability of φ, given the joint state of x and y. This fact increases the difficulty of systematic design of machines that rely on statistical information. The nonmonotonicity of probabilities also seems closely related to recent work in artificial intelligence on nonmonotonic reasoning as a model of defeasible inference. Under the probabilistic interpretation of information, it is easily seen how one location, x, can believe φ in isolation, and how $\neg\varphi$ can be believed by some larger location that includes x as a proper subcomponent. The net effect can be regarded as the overturning of x's belief in φ.

It is also possible to extend informational denotation functions to the probabilistic case. Probabilistic denotation functions would map values into mappings from propositions into the interval $[0,1]$, thereby, given a particular value, providing a distribution of the probabilities of all propositions. In practice, a probability-based denotational analysis is more technically difficult than non-probabilistic analysis, because there is no longer a single *strongest* belief statement that can be made about an agent.

The Localization of Information

The deductive-closure axiom has been viewed as problematic by many AI researchers, who have claimed that it could not possibly apply to agents with limited resources. These researchers, on the whole, have adopted a very concrete view of representation, seeing every ascription $K(X,\varphi)$ as carrying a commitment to the explicit representation of φ as a separate data structure. Clearly, finite agents do not have enough time or space to derive an explicit representation of each known fact.

An immediate consequence of the situated-automata view is that, at any instant, the inputs to the machine, together with its state, implicitly contain the most precise information available to the machine. Any "circuitry" that merely arranges for some component (or location) to contain a value that is a pure function of the inputs and state (computed in negligible time) cannot increase the information content and will, in general, reduce it, since, by mapping distinct information-carrying values to a single value, the result is disjunctive in content and what was heretofore distinct is now blurred. In particular, "inference rules" that syntactically map data structures to other data structures fall into this category; they

cannot be regarded as increasing the system's information.

What, then, is the role of such mappings and how do resource limitations enter the picture? Rather than focus on what information is known, our attention should be focused on which part of the machine knows what and when. A composite agent x may know proposition φ even if none of its subcomponents carries any such knowledge of φ. Much of what is conventionally regarded as "inference" may be thought of instead as localization of information, that is, causing sublocations of the agent to know what only the agent as a whole knew previously. This is crucial for agents that act intelligently upon the world by localizing information at effectors.

We have strong intuitions, too, that resource bounds limit our ability to introspect, and yet the positive and negative introspection axioms seem to imply that the correlational definition of knowledge gives an agent unlimited powers of introspection. Here, too, careful attention to which part of the machine has the knowledge is crucial. The axiom asserts that, if x knows φ, then x as a totality knows that he knows it. It does not follow that any proper subpart of x carries the information that all of x knows φ. We can define a related property of *proper* introspection: an agent x is properly introspective if, whenever x knows φ, there is a proper subcomponent of x that knows that x knows φ. Clearly, finite agents have only finite descending chains of proper subcomponents and thus cannot satisfy iterated proper introspection.

Conclusions

Two of the challenging problems encountered in robot analysis and synthesis are (1) how to relate the representations used in perceptual processing to the representations employed in in higher-level reasoning and planning, and (2) how to process information from the environment in real time. The framework presented in this paper appears to offer advantages in both areas. Since the attribution of *knowledge* is based on objective behavioral properties of the machine, the propositional content of different representational structures can be expressed in a common framework. Furthermore, since the attribution of *content* does not depend on general-purpose deduction mechanisms, much of the system's permanent knowledge can be compiled into the structure of the machine, thereby decreasing the amount of computation required at run time.

Acknowledgments

We have profited greatly from discussions with David Chapman as well as from comments by Fernando Pereira on a previous draft.

References

[1] David Chapman. Personal communication. 1986.

[2] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.

[3] Chris Goad. Special purpose automatic programming for 3d model-based vision. In *Proceedings of the Image Understanding Workshop*, pages 94–104, Arlington, Virginia, 1983.

[4] Mike Gordon. *A Machine Oriented Formulation of Higher Order Logic.* Technical Report 68, University of Cambridge Computer Laboratory, Cambridge, England, 1985.

[5] Mike Gordon. *Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware.* Technical Report 77, University of Cambridge Computer Laboratory, Cambridge, England, 1985.

[6] Joseph Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the 3rd ACM Conference on Principles of Distributed Computing*, pages 50–61, 1984. A revised version appears as IBM RJ 4421, 1984.

[7] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1966.

[8] Jaako Hintikka. *Knowledge and Belief.* Cornell University Press, Ithaca, New York, 1962.

[9] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co., London, England, 1968.

[10] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, Cambridge, Massachusetts, 1984.

[11] Leslie Pack Kaelbling. *Rex Programmer's Manual.* Technical Report 381, Artificial Intelligence Center, SRI International, Menlo Park, California, 1986.

[12] Kurt Konolige. *A Deduction Model of Belief and its Logics.* Technical Report 326, Artificial Intelligence Center, SRI International, Menlo Park, California, 1984.

[13] Saul Kripke. Semantical analysis of modal logic. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[14] Daniel Lehmann. Knowledge, common knowledge, and related puzzles. In *Proceedings of the 3rd ACM Conference on Principles of Distributed Computing*, pages 62–67, 1984.

[15] Hector J. Levesque. A logic of implicit and explicit belief. In *Proceedings of the National Conference on Artificial Intelligence*, pages 198–202, 1984.

[16] Zohar Manna and Amir Pnueli. Verification of concurrent programs: the temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, Academic Press, London, England, 1981.

[17] John McCarthy, P. W. Abrams, D. J. Edwards, T. P. Hart, and M. I. Levin. *Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 2 edition, 1965.

[18] Robert C. Moore. A formal theory of knowledge and action. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*, Ablex Publishing Companey, Norwood, New Jersey, 1985.

[19] Nils J. Nilsson. *Shakey the Robot.* Technical Report 323, Artificial Intelligence Center, SRI International, Menlo Park, California, 1984.

[20] Stanley J. Rosenschein. Formal theories of knowledge in ai and robotics. *New Generation Computing*, 3(4):345–357, 1985.

[21] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, England, 1985.