THE SYNTHESIS OF DIGITAL MACHINES WITH PROVABLE EPISTEMIC PROPERTIES

Stanley J. Rosenschein
Artificial Intelligence Center
SRI International
Center for the Study of
Language and Information
Stanford University

Leslie Pack Kaelbling
Artificial Intelligence Center
SRI International
Center for the Study of
Language and Information
Stanford University

ABSTRACT

Researchers using epistemic logic as a formal framework for studying knowledge properties of 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. By defining $K(x, \varphi)$, 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. 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 multi-agent system and reasoning about the flow of information among them. We also introduce Rex, a language for recursively computing machine descriptions, and illustrate how it can be used to construct machines with provable knowledge properties.

Introduction

Many important computer applications, such as process control, avionics, robotics, and artificial intelligence, involve the design of hardware and software that are part of a larger system embedded in a physical environment. In typical applications of this kind, the computer's principal task is to track and react to conditions in the environment. As more open-ended environments are considered and as the conditions to be recognized and the responses to be generated become more complex, the designer's task becomes correspondingly more difficult. 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 of the work on formalizing knowledge that has been done in philosophy [3,7], theoretical computer science [2], and AI [12,9,6] has focused on abstract properties and has not been concerned with concrete physical or computational interpretations of the central concept of "knowledge." 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 computationally derived from other sentences in the knowledge base. One important advantage of this approach is the ease with which the designer can attribute propositional interpretations to the machine's states. However, there are disadvantages as well, notably in the area of computational complexity.

The situated-automata approach [13], in contrast, takes as its point of departure a concrete computational model for epistemic logic that is compatible with, but does not depend on, viewing the system as manipulating sentences of a logic. 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 x is said to know a proposition φ (written $K(x, \varphi)$) in a situation where its internal state is v if φ holds in all possible situations in which x is in state v. This definition of knowledge satisfies the axioms of modal system S5, including consequential 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 complex applications ordinarily take on too many states to enumerate explicitly, these machines 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 such hierarchically constructed machines. In particular, we use the situated-automata model of knowledge to analyze synchronous digital machines by viewing their components as elements of a multi-agent system and reasoning about the flow of information among these components.

On the practical level, the situated-automata approach has led to the development of Rex, a set of tools for constructing complex machines with rigorously definable epistemic properties. Instead of writing a program that defines the target machine directly, the designer writes a program that, when run, computes a logical description of the machine. This description is

then effectively realized either 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 need not be real-time since it is not itself intended to be coupled to the physical environment.

Theoretical Background

A theory of intelligent embedded systems must be capable of describing how states of the system encode information about other parts of the world over time. Thus the logic must, at the very least, deal with processes, their states, time, and knowledge. 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, i.e., 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. Before defining the semantics, we specify the formal language.

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),
	$*, \circ$	(term operators).

Next, we give the formation rules:

Terms:
1. If e is a *process* (resp. *value*) *constant*, then e is a *process* (resp. *value*) *term*.
2. If e_1, e_2 are *process* (resp. *value*) *terms*, then so is $[e_1 \mid e_2]$ (resp. $\langle e_1 \mid e_2 \rangle$).
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: $\Diamond\varphi = \neg\ Box x \neg\varphi$, 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 lower case 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. Possibility is modeled using 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. Processes occupy collections of atomic locations; these collections vary with time and possible world. In order to conveniently name subprocesses, we structure space and state by closing the set of locations and the set of states under *pairing* operations. Formally, 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 $(x, y) \in pairs(A)$.

Models: $\mathcal{M} = (W, T, L, Q, q, I = (I_P, I_A, I_C, I_F))$, 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. Q is a nonempty set of *atomic states*.
5. $q : L \times W \times T \rightarrow Q$ 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. $I_C : C \rightarrow pairs(Q)$ interprets value constants as value structures.
9. $I_F : F \rightarrow pairs(Q) \rightarrow pairs(Q)$ interprets function symbols as functions on value structures.

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

$$\begin{aligned} \hat{q}(\mathit{nil}_L, w, t) &= \mathit{nil}_Q, \\ \hat{q}(\ell, w, t) &= q(w, t, \ell) \text{ for } \ell \in L, \\ \hat{q}((s_1, s_2), w, t) &= (\hat{q}(s_1, w, t), \hat{q}(s_2, w, t)) \end{aligned}$$

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

1. $[\![a]\!]_t^w = I_A(a)(w,t)$ if a is a *process constant.*
2. $[\![c]\!]_t^w = I_C(a)$ if c is a *value constant.*
3. $[\![[x \mid y]]\!]_t^w = ([\![x]\!]_t^w, [\![y]\!]_t^w)$.
4. $[\![\langle u \mid v \rangle]\!]_t^w = ([\![u]\!]_t^w, [\![v]\!]_t^w)$.
5. $[\![*x]\!]_t^w = \hat{q}([\![x]\!]_t^w, w, t)$, where
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 M and a world-time pair w, t (again we supress 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 $\hat{q}([\![y]\!]_t^w, w, 0) = I_C(c)$ and $\hat{q}([\![y]\!]_t^w, w, t+1) = \hat{q}([\![x]\!]_t^w, w, t)$
4. $w, t \models \Pi_f(x, y)$ if $\hat{q}([\![y]\!]_t^w, w, t) = I_F(f)(\hat{q}([\![x]\!]_t^w, w, t))$
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', t' such that $\hat{q}([\![x]\!]_{t'}^{w'}, w', t') = \hat{q}([\![x]\!]_t^w, w, t)$.

A model M is said to simply *satisfy* a formula iff $M, w, t \models \varphi$ 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 $\Gamma \models \varphi$) iff every model that satisfies each sentence of Γ also satisfies φ.

We do not present a axiomatic treatment of the logic; the interested reader is referred to standard treatments of modal logic [4], logics of time and knowledge [10,8], and their application to AI [12,13]. 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 these are listed here.

1. Theorems of propositional logic and temporal logic [10,1].

2. $\models K(x,\varphi) \rightarrow \varphi$ (truth).
3. $\models K(x,\varphi \rightarrow \psi) \rightarrow (K(x,\varphi) \rightarrow K(x,\psi)))$ (consequential closure).
4. $\models K(x,\varphi) \rightarrow K(x,K(x,\varphi))$ (positive introspection).
5. $\models \neg K(x,\varphi) \rightarrow K(x,\neg K(x,\varphi))$ (negative introspection).
6. $\models *X = v \rightarrow K(X, *X = v)$ (self-awareness).
7. $\models \Box_W \varphi \rightarrow \varphi$
8. $\models \Box_T \varphi \rightarrow \varphi$
9. $\models \Delta_c(x,y) \rightarrow \Box_T(start \rightarrow *y = c \wedge \circ * y = *x)$
10. $\models \Pi_f(x,y) \rightarrow \Box_T(*y = f(*x))$
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. $\varphi \models \bigcirc_T \varphi$ (succedent necessitation).

Modeling Machines in the Logic

Physical processes (of which the processes in the logic are idealizations) can be described in many different 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 form yet another class, specifying the information carried by processes. These descriptions may be interrelated; for instance, a structural constraint may entail certain behavioral properties, which, in turn, have epistemic correlates.

We use the logic to model a *machine* as a collection of (possibly complex) processes related by various constraints of the types discussed above. The physical components of a digital computer may be modeled as processes. When we wish to be concrete, we refer to these processes as *storage locations*. 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 made up of primitive contraints corresponding to the basic component types. Pure functional machines (e.g., logic gates), are modeled in the logic by function-element predicates Π_f, and delay components are modeled by delay-element predicates, Δ_c. The component modeled by Π_f "instantaneously" computes the primitive function f; the delay component modeled by Δ_c initially emits the constant c, followed by the stream of its input values, displaced in time by one unit.

Rex : A Framework for Hierarchical Machine Specification

Rex is a language for the hierarchical specification of complex machines made up of the primitive types discussed in the previous section. From the Rex specification of a machine, a low-level machine description is computed. 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 (software, 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 [11] to include forms that incrementally calculate the low level description of a machine. This section will provide a formal description of a simple subset of Rex, which, although it has the full power of Rex, is somewhat more tedious to program in than the full language. A more practical introduction to the full language is given in a separate reference manual [5].

Rex extends Lisp through the addition of a number of forms which compute machine descriptions. There are two types of Lisp value that we will use to represent these machine descriptions. They are *constraints* and *constrained storage terms.* A constraint represents a conjunction of wffs 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 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 the same value as the other; behaviorally equivalent storage locations may be conveniently realized as the same physical storage location. Constrained storage terms are made up of a distinguished storage term and a wff which specifies the behavior of the location named by that storage term in terms of other storage locations.

We will give 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 takes *st*, a storage term, and *c*, a constraint, and returns the constrained storage term made up 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. *init* must be 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 is expressed in the logic by $\Delta_{init}(next, result)$.

(`make-pi` *fcn* (arg_1 ... arg_n) *result*) This function creates a wff that specifies that the con-

tents of the storage location denoted by *result* be the result of applying *fcn* to the contents of the storage locations of $arg_1 \ldots arg_n$. 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 are 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 which 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 a constraint which 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 (make-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 named by the name of the function they compute with an 'm' appended to the end.

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 have be able to dynamically control the specification of machines at compile time. We use the Lisp form `if` to create machine specifications which are conditional on the values of Lisp expressions at compile time. Arguments which are not constrained storage terms, are referred to as *value parameters*, and are also used in the compile-time control of machine specification.

Complex functions returning constraints or constrained storage terms may be built up out of the Rex forms, and defined using the Lisp **defun** form. Once such functions are defined, a low-level machine description is calculated in two steps. The first step consists of evaluating a Rex form or function that returns a constraint. This constraint is a machine description, but is not yet in conveniently usable form, since there are typically a large number of equivalence wffs, making it difficult to see 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 non-canonical 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 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 names for non-canonical 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 (since it was started) had one, two, or three as its input. Following are the function definitions that make up the specification of the machine.

```
(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))
```

The constant function returns a wff term that requires the storage location of const to always contain the value value. The memberm function is a recursive specification of the standard member function. length is an integer specifying the length of list, which is list of constrained storage terms, and item is a simple constrained storage term. The storage location of the result is constraint to always will contain a 1 if the the contents of item is 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 runtime computation structure. Note also the use of == to structurally decompose list 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

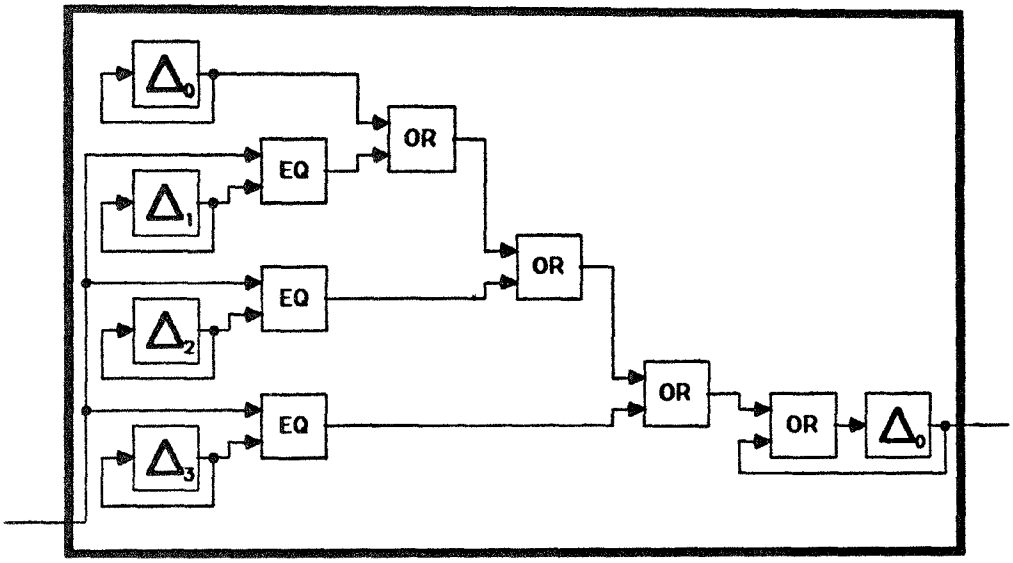
```
(ever-a-one-two-three (storage 'input) (storage 'output))
```

which will calculate a non-canonical constraint. The left column of the following figure contains a listing of part of the non-canonical constraint. The symbols with numeric suffixes were generated by gensym. Since there are 76 equational wffs, we only exhibit a few of them. The right column contains the entire constraint after it has been canonicalized.

```
((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))
```

This can be interpreted as a linear description of the wiring diagram of the "circuit" diagrammed below.



Example

In this section we present the Rex specification of a robot that intermittently senses the location of a moving object in its environment and tries to keep track of whether the object is within *shouting distance* (a certain radius) of the robot. Due to the degradation of information over time and motion of the robot, sometimes the machine will *know* that the object is within shouting distance, sometimes it will *know* that the object is *not* within shouting distance, and sometimes it will not know either proposition. We characterize the epistemic properties of the outputs of this machine in terms of the epistemic properties of its inputs and sketch a

proof of this characterization.

Description of the shoutm Machine

The machine has three inputs, which we will refer to as x, y, and action. The values of 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 x and y are equal to the distiguished constant BOTTOM. The values of action encode the robot's last action; it can take on four possible values: MOVE means that the robot moved one unit in the direction it was facing; RTURN means that the robot turned 90 degrees to the right; LTURN means that the robot turned 90 degrees to the left; NOOP means that the robot did nothing.

The machine has two output lines, K-shout-dist and K-not-shout-dist. We show that K-shout-dist *knows* that the object is within shouting distance whenever its value is 1, and that 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 x, y, and action, on the one hand, and K-shout-dist and K-not-shout-dist, on the other, by introducing complex intermediate locations, 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 last 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, since 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 K-shout-dist-vec, a vector of boolean values, the *i*th of which has the value 1 if the information in the *i*th element of the queue entails that the object is within shouting distance. Similarly, the queue is mapped into 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 and their connection to queue not only by size, but 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*, i.e., 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:

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

where $x, y \in \mathbf{N}$ and $a \in \{\texttt{MOVE}, \texttt{LTURN}, \texttt{RTURN}, \texttt{NOOP}\}$. 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. $\mathit{doing}(a)$ expresses the proposition that the robot is doing the action referred to by a. We assume that Γ contains, among other things, axioms relating *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 $\Gamma, \texttt{queue-with-transform}(n, [X, Y], A, [E_1, \ldots, E_n]), \mathit{InputAxioms}(X, Y, A) \models$

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

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 depending on the 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 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 "multiple 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,

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

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

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

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

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

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

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

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 a fact about the epistemic properties of machines that is an instance of the more general fact that all agents know necessary truths.

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

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 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:

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

Application

The software tools and analytic techniques described in this paper are currently being applied to SRI's mobile robot. An implementation of Rex exists that generates sequential code for controlling the robot. Since the machine descriptions computed by Rex are similar to circuit diagrams, more direct realizations in hardware are being considered. Two of the challenges in robot analysis and synthesis are: (1) how to relate the representations used in perceptual processing to the representations involved 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 more easily compared. Furthermore, since the attribution of content does not depend on general-purpose deduction mechanisms, much of the permanent knowledge of the system can be compiled into the structure of the machine, decreasing the amount of computation required at runtime.

Acknowledgments

This work was supported in part by a gift from the Systems Development Foundation, in part by FMC Corporation under contract 147466 (SRI Project 7390), and in part by General Motors Research Laboratories under contract 50-13 (SRI Project 8662).

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

References

[1] Gabbay, Dov, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. "On the Temporal Analysis of Fairness." Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, 1980, pp. 163-173.

[2] Halpern, Joseph and Y.O. Moses. "Knowledge and Common Knowledge in a Distributed Environment." Proceedings of the 3rd ACM Conference on Principles of Distributed Computing, 1984, pp. 50-61; a revised version appears as IBM RJ 4421, 1984.

[3] Hintikka, J. *Knowledge and Belief.* Cornell University Press, Ithaca, 1962.

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

[5] Kaelbling, Leslie Pack. *Programming in Rex.* Technical Note, Artificial Intelligence Center, SRI International, Menlo Park, CA, (forthcoming.)

[6] Konolige, Kurt. *A Deduction Model of Belief and its Logics.* Technical Note No. 326, Artificial Intelligence Center, SRI International, Menlo Park, CA, August, 1984.

[7] Kripke, Saul. "Semantical Analysis of Modal Logic." *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik* 9, 1963, pp. 67-96.

[8] Lehmann, Daniel. "Knowledge, Common Knowledge and Related Puzzles." Proceedings of the 3rd Annual ACM Conference on Principles of Distributed Computing, 1984, pp. 62-67.

[9] Levesque, Hector J. "A Logic of Implicit and Explicit Belief." Proceedings of the National Conference on Artificial Intelligence, 1984, pp. 198-202.

[10] Manna, Zohar and Amir Pnueli, "Verification of Concurrent Programs: the Temporal Framework," *The Correctness Problem in Computer Science* (R. S. Boyer and J. S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London, 1981.

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

[12] Moore, Robert C. "A Formal Theory of Knowledge and Action." In *Formal Theories of the Commonsense World*, Jerry R. Hobbs and Robert C. Moore (eds.), Ablex Publishing Company, Norwood, New Jersey, 1985.

[13] Rosenschein, Stanley J. "Formal Theories of Knowledge in AI and Robotics." *New Generation Computing* Vol. 3, No. 4 (in press). Ohmsha, Ltd. Tokyo, Japan.