massachusetts institute of technology — artificial intelligence laboratory

Generating Communications Systems Through Shared Context

Jacob Beal

AI Technical Report 2002-002 January 2002

© 2002 massachusetts institute of technology, cambridge, ma 02139 usa — www.ai.mit.edu

Generating Communication Systems Through Shared Context

by

Jacob Beal

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 31, 2002

© Massachusetts Institute of Technology 2002. All rights reserved.

Certified by: Gerald Jay Sussman
Matsushita Professor of Electrical Engineering, MIT
Thesis Supervisor

Accepted by: Arthur C. Smith
Chairman, Department Committee on Graduate Students

Generating Communication Systems Through Shared Context

by

Jacob Beal

Submitted to the Department of Electrical Engineering and Computer Science on January 31, 2002, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

In a distributed model of intelligence, peer components need to communicate with one another. I present a system which enables two agents connected by a thick twisted bundle of wires to bootstrap a simple communication system from observations of a shared environment. The agents learn a large vocabulary of symbols, as well as inflections on those symbols which allow thematic role-frames to be transmitted. Language acquisition time is rapid and linear in the number of symbols and inflections. The final communication system is robust and performance degrades gradually in the face of problems.

Thesis Supervisor: Gerald Jay Sussman
Title: Matsushita Professor of Electrical Engineering, MIT

Acknowledgments

Particular note should be given to the help from several people. Gerry Sussman started me thinking about the problem and pointed me in this direction, as well as providing direction, guidance, and good advice. Patrick Winston was invaluable in forcing me to think about the reasons behind my mechanisms. My research is part of a continuing effort started by Yip and Sussman to build a "TTL databook for the mind" — a compatible system of modules that capture aspects of mental activity and can be combined to build ever more powerful systems. Thanks also to Catherine Havasi for hacking some of the early code with me and being a needling presence to keep me from slacking off.

Contents

1 Introduction **7**
1.1 Organization of Thesis 8

2 Problem Domain **9**
2.1 System Model 9
2.1.1 Encoding View 11
2.1.2 Experimental Specifics 12
2.2 Restriction on Types of Solution 12

3 Algorithm **16**
3.1 Concept 16
3.2 Implementation 17

4 Experimental Results **22**
4.1 Convergence Time 23
4.2 Channel Capacity 26
4.3 Performance Degradation 26
4.4 Parameter Variation 28
4.5 Dissimilar Features 30

5 Contributions **32**
5.1 Vision 33

A Scheme Code **36**
A.1 Algorithm 36
A.2 Testbed 42

List of Figures

2.1 The agents labelled A and B are interconnected by *comm lines* — a bundle of wires with an arbitrary and unknown permutation. The agents also share some *feature lines* with the outside world, again with unknown permutations. 10
2.2 The communication system viewed as an encoding problem: the goal is to train the encoder and decoder to correctly communicate M from input to output despite the unknown permutation between them. 12
2.3 During a training cycle, the feature lines of both units are driven. Each agent attempts to learn from the comm lines driven by the other agent. 13
2.4 During a test cycle, the feature lines of one unit, say A, are driven and the feature lines of the other unit are observed. The test is scored by number of mistakes in B's reproduction of A's feature lines. 14

4.1 Number of shared symbols versus elapsed time for 50 nouns and 20 verbs. Dotted line is theoretical estimate S(t), solid line is experimental data. 24
4.2 Number of shared inflections versus elapsed time for 4 noun inflections and 1 verb inflection, in a system with 50 nouns and 20 verbs. The dotted line is theoretical estimate I(t), beginning with cycle 230, where S(t) predicts half the vocabulary to be learned. The solid line is experimental data. 25

4.3 Number of comm lines versus transmission robustness. Horizontal axis is n_w from 100 to 100,000. Vertical axis shows the symbols and inflections correctly received per symbol and inflection transmitted (spurious receptions count against this as well) over the course of 200 test cycles on a system trained with 50 nouns, 20 verbs and 4 noun-roles, $n_{wps} = 20$, $p_s = 0.8$. Accuracy degrades smoothly with decreased channel capacity. 27
4.4 Variation in performance as each parameter is varied. For each graph, the horizontal axis shows the value of the parameter being varied and the vertical axis shows the fraction of symbols and inflections correctly received per symbol and inflection transmitted. Measurements are the average values over the course of 10 runs of 200 test cycles, as in 4.3. For each run of test cycles, the systems were trained with 50 nouns, 20 verbs and 4 noun-roles, with base parameter values $p_s = 0.8$, $r_i = 0.05$, $t_c = 4$, $t_p = 6$, $w_m = 20$, $n_{wps} = 100$, and $n_w = 10000$. All parameters in the system can tolerate small variations without serious degradation in performance. This figure shows variation in p_s and r_i. Figure 4.5 shows variation in t_c, t_p, w_m, and n_{wps} 28
4.5 Variation in performance as each parameter is varied, as in Figure 4.4, for t_c, t_p, w_m, and n_{wps} 29

Chapter 1

Introduction

Neuroscience has postulated that the brain has many "organs" — internal subdivisions which specialize in one area. If we accept this view, then we need some sort of mechanism to interface these components. The design of this mechanism is limited by the hardware which the brain is constructed out of, as well as the size of the blueprints specifying how it is built. Neurons, as hardware, are relatively slow and imprecise devices, but they are very cheap, and it's easy to throw a lot of them at a problem in parallel. Our DNA is only about 1 gigabyte, too small to encode the full complexity of interfaces between all of the different components.

I approached this design problem from a hardware hacking point of view, with the question, "If I were designing the human brain, how would I build this interface?" It needs to be self-configuring, to beat the limited blueprints problem, and it needs to learn quickly. On the other hand, hardware is very cheap, and I can design in a domain with a huge number of interface wires between two components.

I have developed an algorithm which, while working under restrictions inspired by human minds, bootstraps communications solely from shared experience and I present it here as an existence proof and a tool for thinking about how a brain might be composed out of independent parts that learn to communicate with each other: it is possible for two agents to rapidly construct a language which enables them to communicate robustly.

Although this dissertation is about the problem primarily in a language framework, the resultant algorithm is applicable to a much wider domain of problems, including multi-agent robotics, chip-to-chip communication, and networking protocols.

1.1 Organization of Thesis

Section 2: Problem Domain Specification and justification of the problem domain in which my system operates.
Section 3: Algorithm Conceptual description of the communications bootstrapping system, as well as its algorithmic implementation.
Section 4: Results Experimental results demonstrating that the system is functional, fast, and robust.

Chapter 2

Problem Domain

In this section, I will develop the problem to be solved, along with the motivation for choosing to structure the problem as I have.

Briefly, the system consists of two agents which must learn to talk to one another, connected by a large twisted bundle of wires. There is a scene presented to either one or both agents. The system is judged to be successful when presenting arbitrary scenes to one agent consistently results in the other agent describing the scenes correctly.

The design by which the agents are constructed is limited as well. Agents must be constructed of simple, bidirectional components. These components should be independently acting, and be deployed in a simply structured arrangement requiring only shallow computational processes.

In the subsequent sections, I expand on this brief treatment:

2.1 System Model

The basis of the system is the two agents which are to learn to communicate with each other. There are only two agents because, in this work, I am restricting myself to the problem of mere communication, rather than any questions of cooperation or how to get useful work out of the system: it is enough that information is reproduced from one to the other. This system, however, can serve as a mechanism on which more abstract systems that do address those questions, like Hearn's work[1] on the implementation of Minsky's Society of Mind,[6] can be instantiated.

The form of this information is a set of ordered pairs, where the first variable in the ordered pair can vary widely, and the second variable

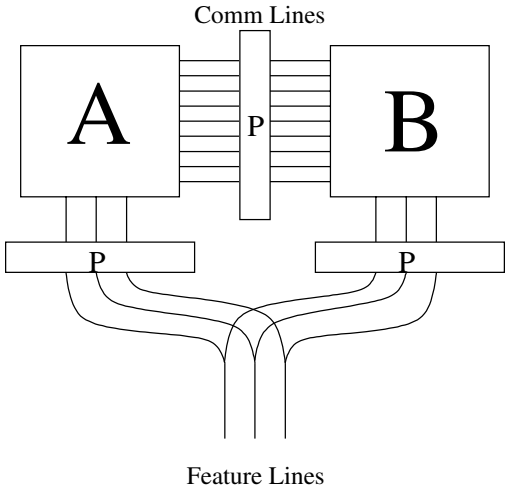


Figure 2.1: The agents labelled A and B are interconnected by *comm lines* — a bundle of wires with an arbitrary and unknown permutation. The agents also share some *feature lines* with the outside world, again with unknown permutations.

is restricted to a small number of values. In the experiments which I conducted, I thought of these sets as thematic role-frame descriptions of the world (e.g. $\{(bob, subject)(jim, object)(kick, predicate)\}$). It is important to note, however, that this is merely an interpretation of the information being conveyed which has nothing to do with my algorithm. In further discussion, I shall refer to the first, many-valued variable as the *symbol* and the second, few-valued variable as the *inflection*.

The two agents are presented with this information via *feature lines* — a bundle of wires with an unknown permutation, connected to the outside world. Each feature line represents a symbol and carries an inflection as the value on the line. If no inflection is specified, then the feature line may be either "driven", representing an uninflected symbol, or "undriven", representing a symbol not in use. A feature line may not, however, carry more than one inflection, so any given symbol can appear in at most one ordered pair.

The two agents are able to communicate information to each other via a set of *comm lines*. There are a very large number of comm lines, and there is an unknown permutation on them as well.[1]

[1] In the experiments I performed, a stronger limitation than unknown permutation was used: no ordering exists on the lines.

The comm lines have four states: 1,-1,0, and X. When undriven, the line reads as 0. Information is sent over the line by driving it to 1 or -1, and if the line is being driven to both 1 and -1, it reads as a conflict — X.

Metaphorically, the comm lines are a nerve bundle connecting two "organs" of the brain and the feature lines are hardware representations of perceptual features of the outside world. The actual wires in the bundles might be arbitrarily twisted and rearranged between the two ends of the system, so we add the unknown permutation to each bundle to model this effect and prevent any implicit sharing of ordering information between the agents.

Each agent can read and drive both comm and feature lines, but agents are constrained to a synchronous schedule: each cycle of the system has a "talk" phase and a "listen" phase. Agents can read lines at any time, but can only drive comm lines during the talk phase and feature lines during the listen phase. At the end of the talk phase, for symmetry breaking purposes one agent is randomly selected to have spoken first. The agent which spoke first can read the results of both agents speaking in its listen phase, while the one which spoke second reads only what the first agent spoke.

The task posed for this system, then, is for each agent to learn how to convey the ordered pairs presented on its feature lines to the other agent, which demonstrates its understanding by presenting those ordered pairs on its own feature lines.

The system must learn to perform this task in an unsupervised manner: while the system can be continuously monitored for performance, to feed that information back into the system would establish an implicit channel of communication between the two systems, defeating the purpose of the exercise.

2.1.1 Encoding View

Communication in this system may be viewed more abstractly as an encoding problem. The set of possible messages M are all sets of ordered pairs (A_1, A_2), where A_1 has a large range and A_2 has a small range, and A_1 appears in at most one ordered pair in the set. The system's problem is to find an encoding system such that any M can be transmitted in a single clock cycle and reconstructed by the decoder.

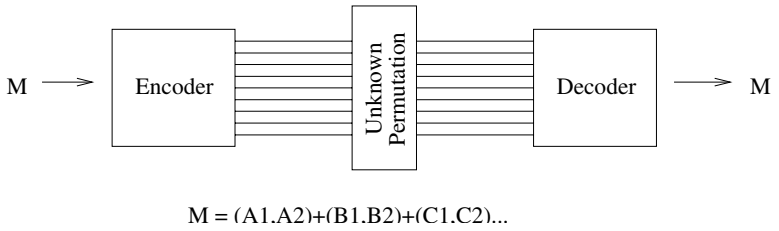


Figure 2.2: The communication system viewed as an encoding problem: the goal is to train the encoder and decoder to correctly communicate M from input to output despite the unknown permutation between them.

2.1.2 Experimental Specifics

In the experiments conducted here, I used a set of 10,000 comm lines, and presented thematic role-frames on the feature lines. The symbols of feature lines are things or actions and the inflections driven on them are roles. So typical symbols might be **Bob**, **Mary**, or **push**, and typical inflections might be **subject**, **object**, or **predicate**. The set of feature lines is of undefined size — the agents have no knowledge of what feature names or roles exist until they encounter them in practice.

The system is run with two types of steps — training cycles and test cycles. In a training cycle, the data on the feature lines is sent to both agents. In a test cycle, one agent is randomly selected to receive input from the feature lines, while the other receives no input. Performance may then be evaluated on the basis of how well the output of the agent receiving no input matches the values on the feature lines.

2.2 Restriction on Types of Solution

I am interested not merely in solving the problem of creating a communication system, but in solving it in a way inspired by the human mind. Thus, I placed a number of restrictions on the design of the agents, making the problem deliberately harder such that the solutions will better elucidate the problems of intelligence. The restrictions fall into two general categories: parsimony and robustness.

Parsimony is inspired by the tendency of evolved systems to waste little in the way of resources, and by the limited storage space in DNA. Thus, the agents must be constructed of simple, replicated structures — i.e. structures which take little information to describe. These

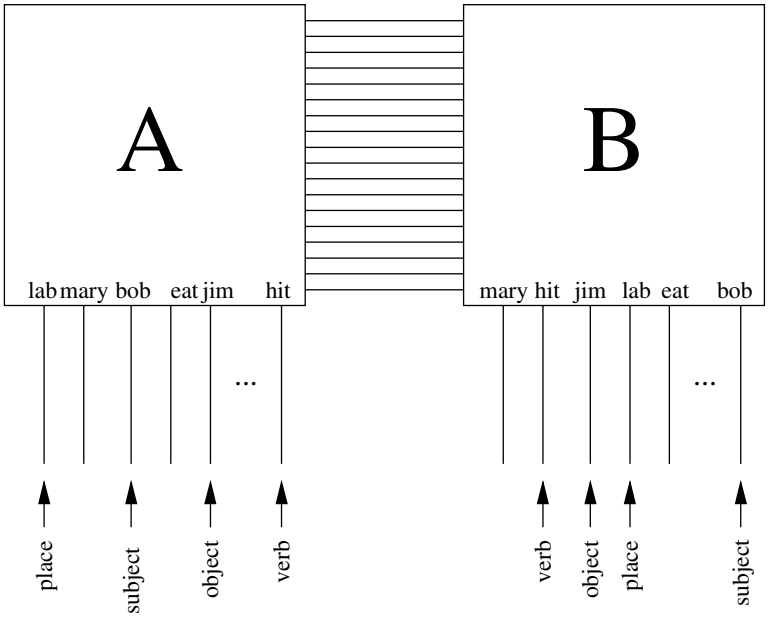


Figure 2.3: During a training cycle, the feature lines of both units are driven. Each agent attempts to learn from the comm lines driven by the other agent.

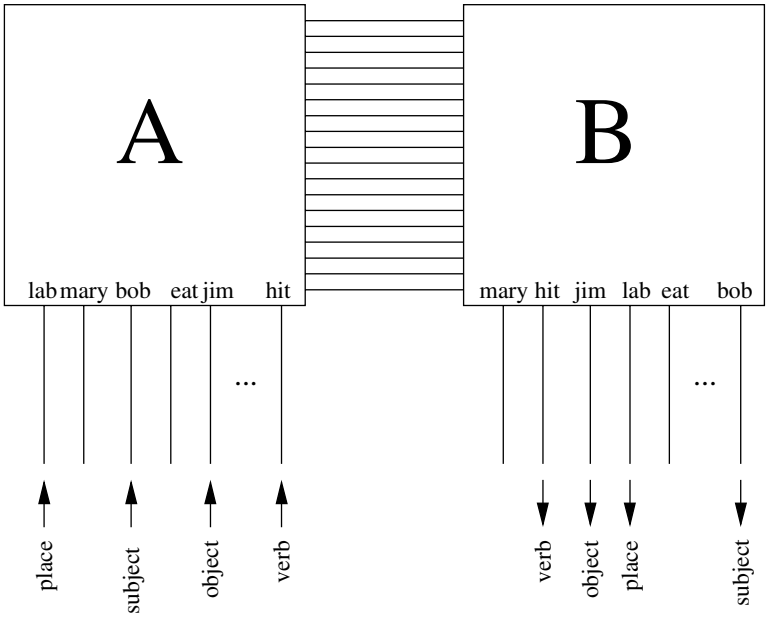


Figure 2.4: During a test cycle, the feature lines of one unit, say A, are driven and the feature lines of the other unit are observed. The test is scored by number of mistakes in B's reproduction of A's feature lines.

structures should start fairly homogeneous and develop rapidly when exposed to stimulus. Each individual element should be relatively simple and act bidirectionally — if possible, we want to learn things only once, and not have to use different hardware for speaking and listening. This also allows us to avoid the problem of maintaining correlation among the different paths of use of a relation.

Robustness is a restriction inspired by the limitations of neural hardware: it is slow, noisy, and prone to component death, yet it operates very reliably in biological systems. Likewise, the robustness requirement means we need to create reliable hardware from unreliable components. The hardware we construct should be able to survive the loss of components, and significant noise in the system. Thus there should be no critical portions of the agent's hardware: interference or destruction of any part of the agent should not cripple its action. Neural hardware is also very slow compared to the speed at which we make decisions, so the hardware we build must be fast, even though its individual component are slow. Thus computational processes should be very shallow — chains of sequential actions should be short, and recursion very limited. Fortunately, hardware is also very cheap, so we are allowed to throw a lot of parallel parts at the problem.

Chapter 3

Algorithm

3.1 Concept

The key idea driving the system design is that sparseness makes it easy to separate stimuli. Given the vast number of comm lines, if we assign small subsets randomly to each symbol, then we may expect little interference between symbols, and an easy analysis job for an algorithm trying to invert the mapping — an idea which has been capitalized on in the past, as in Mooers' zatocoding system which organized information by keywords encoded as punches in index cards,[7] or in Minsky's K-Lines theory of memory, where agents are assigned tags allowing reconstruction of partial "states of mind"[5]

I built agents to solve this problem out of bidirectional constraints. A bidirectional constraint is a computational device which embodies a relation — for example, the connection between English pluralization and linguistic features captured by the constraints in Yip and Sussman's work on one-shot learning.[10] When connected between two systems, the constraint attempts to bring the values of those systems into a state consistent with its relation. Bidirectionality means that the constraint mutates both systems rather than choosing one to be brought into alignment with the other.

In building this system, I used a slightly simplified version of constraints, modified to make learning simple: When a constraint is stimulated on both ends, it learns. When it is stimulated above some threshold on one end, it asserts a compatible value at the other. When it is not stimulated enough on either end, it is quiescent.

The agents are conceptualized as a bundle of constraint hardware, in two parts. One family of constraints is the vocabulary of symbols:

there is one constraint associated with each feature line, connecting it with a set of comm lines. The constraint begins with a small number of comm lines that it is definitely connected to, and the assumption that it might be also connected to any of the other comm lines. The constraint narrows down the set of comm lines it is connected to by intersection with the set of active comm lines each time it learns. Once it has gone through a certain number of learning cycles, it assumes that it has figured out the correct set. The other family of constraints is much smaller and holds the inflections. These modulate the output of the symbol outputs, and learn by a simple agreement algorithm. As in Kirby's work on language evolution,[2] the assumption that both speaker and listener share the same intent allows the system to leverage mutual understanding from shared observations of the external world.

This system, then, obeys the restrictions imposed by parsimony and robustness. The system starts out with a set of bidirectional constraints produced by a simple homogeneous process, then specializes them as it is exposed to information. Because we are using bidirectional constraints, the same hardware translates from feature lines to comm lines as from comm lines to feature lines. Thus, parsimony is satisfied. Robustness is satisfied by the independence of the constraints from one another, and the fact that information is spread evenly across the comm lines used by any given constraint. Thus, an error in a comm line should affect only the constraints using it, and have little effect on them, while an error in a constraint damages only the expression of a single symbol or inflection. Finally, the constraint system is only two layers deep — symbols, and the inflections modulating them — and the constraint hardware itself can be implemented very shallowly. Thus, the robustness restriction is satisfied as well.

3.2 Implementation

I implemented the system as an algorithm which simulates the operation of bidirectional constraint hardware.

Knowledge in the system is represented by two sets of mappings: symbol mappings and inflection mappings — these mappings are the rendering of the constraints. An inflection mapping links a symbol carried on a feature line to a real value between 0 and 1. A symbol mapping links a feature line with two sets of comm lines, designated as *certain* and *uncertain*, and includes an integer designated *certainty*.

These mappings are used symmetrically for production and interpretation of messages. In the "talk" phase, each driven feature line

selects the *certain* comm lines associated via the symbol mapping and drives them with the unary fraction associated with the symbol on the feature line via the inflection mapping. In the "listen" phase, if enough of a feature line's associated comm lines are driven, then the feature line is driven with any inflection mapping symbols within a fixed radius of the unary code on that set of comm lines.

Both types of mappings are generated randomly when a feature or inflection is first encountered, then adjusted based on observations of the other agent's transmissions. These adjustments take place only if an agent spoke second; if it was the first one to speak, then its own transmissions are on the lines as well, inextricably mixed with the transmissions of the second agent, and this would make accurate learning significantly more difficult.

Inflection mappings are adjusted with a very simple agreement algorithm: if the received unary code is significantly different from expected, the code in the mapping is set to the received value. If a unary code matches which should not have, then it is purged and generated anew.

Symbol mappings are slightly more complicated. The first time an agent hears a given symbol spoken by the other agent, it adds every driven comm line to the *uncertain* lines for that symbol. Each time thereafter that it hears the symbol again, it intersects the driven lines with its *uncertain* lines, thereby eliminating lines associated with other symbols. After several iterations of this, it assumes that there is nothing left but lines which should be associated with the symbol, adds the *uncertain* lines to the *certain* lines, and begins to use them for communication. A few more iterations after that, it begins paring down the *certain* lines the same way, so that the two agents can be assured that they have identical mappings for the symbol.

Formal Automaton Description:
Constants:

(Note that the precise values of these constants was
arbitrarily chosen, and the behavior of the
algorithm should be insensitive to small changes.
For example, $p_s = 0.8$ just means ''a pretty good
match'' and $t_c = 4$ means ''a few times'' ---
see Results section for more analysis.)

r_i = 0.05 (Radius of an inflection value)
p_s = 0.8 (Percent stimulus required to match a symbol)
t_c = 4 (Threshold where *uncertain* lines become *certain*)
t_p = 6 (Threshold to prune *certain* lines)
n_w = 10000 (Number of comm lines)
n_{wps} = 100 (Number of comm lines randomly selected for a

new symbol)
w_m = 20 (Minimum number of comm lines per symbol)

Input:
$talk_{in}(F)$
F is a set of (s,i), where s,i are symbols (feature lines)
$listen_{in}(C_i, F, first)$
C_i, $0 < i \leq n_w$, $\in \{1, 0, -1, X\}$ (comm lines)
$first$ is boolean, F as for $talk_{in}$

Output:
$talk_{out}(C_i)$, C_i as for $listen_{in}$
$listen_{out}(F)$, F as for $talk_{in}$

States:
T_s is a set of $x = (x_s, x_c, x_u, x_n)$ where x_s is a symbol, x_c, x_u are sets of $r \in N^+$, $0 < r \leq n_w$, and $x_n \in N^+$ initially $\emptyset$
T_i is a set of $y = (y_i, y_v)$ where y_i is a symbol and $0 \leq y_v \leq 1$ initially $\emptyset$
c is a set of (l, v) where $l \in N^+$, $0 < l \leq n_w$ and $v \in \{1, 0, -1, X\}$, initially $\emptyset$, with the join rules $(l,1) \cup (l,-1) = (l, X)$, and $(l, X) \cup (l, *) = (l, X)$
f is a set of (s,i) where s,i are symbols, initially empty
$talking, listening$ are booleans, initially false

Transitions:
$talk_{in}(F)$
Effect:
$talking$:= true
$\forall\ (s,i) \in F$
if not $\exists\ x \in T_s$ s.t. $x_s = s$
$T_s := T_s \cup (s$,set of n_{wps} random elements,$\emptyset$,0)
if not $\exists\ y \in T_i$ s.t. $y_i = i$
$T_i := T_i \cup (i$,random)
let $x \in T_s$ s.t. $x_s = s$
$y \in T_i$ s.t. $y_i = i$
$v_l \in \{1,-1\}$, $\forall\ l \in x_c$ s.t.
precisely $|x_c| * y_v$ of the set $\{v_l\}$ are 1
$\forall\ l \in x_c$
$c := c \cup (l,\ v_l)$

$talk_{out}(C_i)$
Preconditions:
$\forall\ (l, v) \in c$

```
      C_l = v
    ∀ l s.t. ∀ v, (l, v) ∉ c
      C_l = 0
    talking = true
  Effect:
    talking := false
    c := ∅

listen_in(C_i, F, first)
  Effect:
    listening := true
    ∀ m ∈ T_s
      if not first and ∃ (s, i) ∈ F s.t. s = m_s
        if m_n = 0
          m_n := 1
          m_u := {j | C_j ≠ 0}
        else
          m_n := m_n + 1
          if m_n < t_c
            m_u := m_u ∩ {j | C_j ≠ 0}
          if t_c ≤ m_n < t_p
            m_c := m_c ∪ m_u
            m_u := ∅
          if t_p ≤ m_n
            if |m_c ∩ {j | C_j ≠ 0}| ≥ w_m
              m_c := m_c ∩ {j | C_j ≠ 0}
            else
              T_s := T_s − m
      else
        let x = m_c ∩ {j | C_j ≠ 0}
            u = |{j ∈ x | C_i = 1}|/|{j ∈ x | C_j ∈ {1, −1}}|
          if |x| ≥ |m_c| * p_s
            if ∃ y ∈ T_i s.t. |y_v − u| < r_i
              ∀ y ∈ T_i s.t. |y_v − u| < r_i
                f := f ∪ (m_s, y_i)
            else
              f := f ∪ (m_s, null)
    if not first
      ∀ y ∈ T_i
        if ∃ (s, i) ∈ F s.t. i = y_i and ∃ m ∈ T_s s.t. m_s = s
          let u = |{j ∈ m_c | C_j = 1}|/ |{j ∈ m_c | C_j ∈ {1, −1}}|
            if |u − y_v| > r_i/2 and {j ∈ m_c | C_j = X} = ∅
              y_v := u
        else
          if ∃ (s, i) ∈ F, m ∈ T_s s.t.
              |(|{j ∈ m_c | C_j = 1}|/
```

$|\{j \in m_c | C_j \in \{1, -1\}\}|) - y_v| < r_i$
`and` $\{j \in m_c | C_j = X\} = \emptyset$
y_v `:= random`
$\forall\ y \in T_i$
`if` $\exists\ z \in (T_i - y)$ `s.t.` $|z_v - y_v| < r_i * 2$
T_i `:=` T_i `- (random` $\in \{y, z\}$`)`

$listen_{out}(F)$
`Preconditions:`
F `=` f
listening `= true`
`Effect:`
listening `:= false`
f `:=` $\emptyset$

See Appendix A for Scheme code implementing the algorithm.

Chapter 4

Experimental Results

To test the algorithm, I used a system with an n_w of 10000 comm-lines and a n_{wps} of 100 random wires selected to generate a new symbol mapping.

I trained the system for 1000 cycles, then evaluated its performance over an additional 200 cycles. Each cycle, an example is generated and presented to the system. In the training phase, there is an 80% chance it will be presented to both agents and a 20% chance it will be presented to only one (That is, 80% training cycles, 20% test cycles). During the evaluation phase, the first 100 are presented to the first agent only, and the second 100 are presented to the second agent only. A test is considered successful if the input feature set is exactly reproduced by the listening agent.

The examples input to the feature lines are thematic role frames generated from a set of 50 nouns, 20 verbs, and 4 noun-roles. Each example is randomly generated with 0-2 verbs assigned the "verb" role and 2-4 nouns assigned noun-roles. No noun, verb, or noun-role can appear more than once in an example. A typical scene, then, might be **((approach verb) (jim subject) (shovel instrument) (lab object))**, which corresponds loosely to "Jim approached the lab with the shovel." All told, there are more than 1.2 billion examples which can be generated by the system, so in general an agent will never see a given scene twice.

In a typical run of this system, after about 200 cycles most symbols will have entered the shared vocabulary and can be successfully communicated between the two agents. After about 500 cycles, the set of inflections will have stabilized as well. In the final round of 200 tests, the success rate is usually 100%, although occasionally due to

the stochastic nature of the algorithm, the inflections will not yet have converged by the end of 1000 tests and consequently one or more will not be transmitted correctly.

4.1 Convergence Time

The time needed to develop a shared vocabulary is proportional to the number of symbols in the vocabulary. A symbol is learned when both agents have *certainty* for that symbol greater than t_c. An agent increases *certainty* when it speaks second, which is determined randomly, so we may estimate this as a Poisson process. Thus, we may calculate the expected number of cycles, c, as follows:

$$E(c) = 2t_c \frac{\binom{2t_c}{t_c}}{2^{2t_c}} + \sum_{n=2t_c+1}^{\infty} n \frac{\binom{n-1}{t_c-1}}{2^{n-1}}$$

Evaluating this for $t_c = 4$, we find an expectation of 10.187 uses of a symbol before both *certainty* thresholds are reached.

For these experiments then, with an average of 3 nouns and 1 verb per training cycle, then, we can calculate the expected number of shared symbols S as a function of elapsed cycles t:

$S(t) = n_{nouns} * (1 - P(10.187, t\frac{3}{n_{nouns}}0.8))+$
$n_{verbs} * (1 - P(10.187, t\frac{1}{n_{verbs}}0.8))$

where P is the incomplete gamma function. Since this function is linear in the number of symbols, we see that the time to build a shared vocabulary is linear in the number of symbols. Figure 4.1 shows experimental data confirming this estimate.

Once a shared vocabulary of symbols exists, the algorithm can begin learning inflections. If n_i is the number of inflections to be learned, and r_i is chosen such that $r_i * n_i \leq 0.5$, then we can show that the time to develop a shared set of inflections is $O(n_i)$.

An inflection may be learned any time a symbol is successfully transmitted in a training cycle. This occurs if the new inflection does not conflict with any of the previously learned inflections - that is, if n symbols have already been learned, then it must be the case that for all v_i s.t. $1 \leq i \leq n$, $|v_{n+1} - v_i| < 2r_i$. Since the value of the new symbol, v_{n+1}, is chosen by a uniform random process on the interval $[0, 1]$, the probability p_{n+1} of choosing an acceptable inflection value is no less than $1 - (2r_i * n)$. The n_ith inflection, then, has the least probability of success, $p_{n_i} = 1 - (2r_i * (n_i - 1)) \geq 2r_i$, and p_n is generally bounded below by $2r_i$.

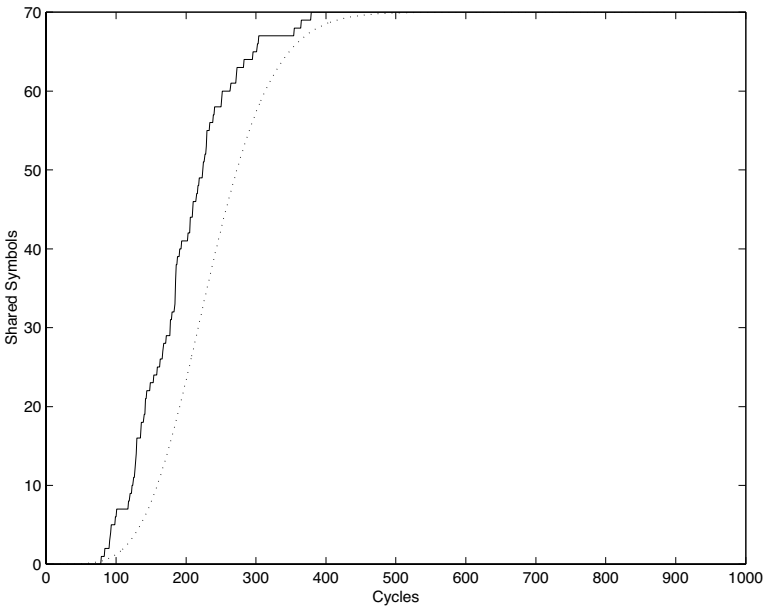


Figure 4.1: Number of shared symbols versus elapsed time for 50 nouns and 20 verbs. Dotted line is theoretical estimate S(t), solid line is experimental data.

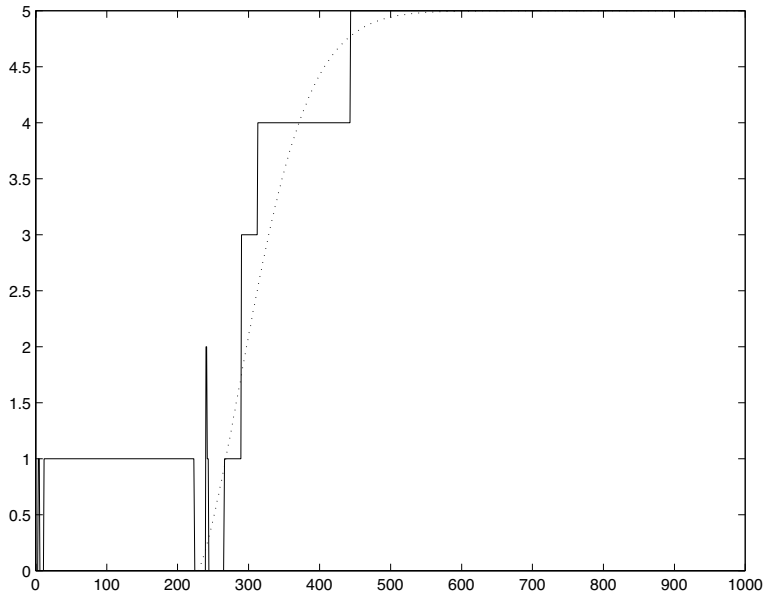


Figure 4.2: Number of shared inflections versus elapsed time for 4 noun inflections and 1 verb inflection, in a system with 50 nouns and 20 verbs. The dotted line is theoretical estimate I(t), beginning with cycle 230, where S(t) predicts half the vocabulary to be learned. The solid line is experimental data.

For these experiments then, we can calculate the expected number of inflections, assuming a shared vocabulary, as a function $I(t)$ of elapsed cycles t. There are expected to be 3 noun inflections and 1 verb inflection per training cycle, so the least frequent inflection is expected to appear at with frequency at least $1/n_i$. Thus, we obtain

$$I(t) = n_i * (1 - P(1, 2r_i t \frac{1}{n_i} 0.8))$$

where P is the incomplete gamma function. Since this function is linear in the number of inflections, we see that the time to build a shared set of inflections is linear in the number of inflections. Figure 4.2 shows experimental data confirming this estimate.

Thus, the algorithm is expected to converge in $O(s+n_i)$ time, where s is the size of the vocabulary and n_i is the number of inflections.

4.2 Channel Capacity

The number of symbols and roles which can be learned without false symbol detection and inflection misinterpretation is dependent on the number of wires n_w, the number of wires per symbol n_{wps}, and the percent stimulus necessary to recognize a symbol p_s.

If we want no combination of symbols to be able to generate a spurious recognition, then each symbol must have at least $n_{wps}(1-p_s)$ wires not used by any other symbol. This means that a vocabulary would have a maximum size of only $\frac{n_w}{n_{wps}(1-p_s)}$. In practice, however, we can assume that only a few symbols are being transmitted simultaneously. If we assume that no more than m symbols will be transmitted at once, then we can conservatively estimate capacity by allowing any two symbols to overlap by no more than $n_{wps} * p_s/m$ wires. Thus any given symbol covers a portion of symbol space with volume:

$$\sum_{i=0}^{n_{wps}(1-\frac{p_s}{m})} \binom{n_{wps}}{i} \binom{n_w - n_{wps}}{i}$$

The whole symbol space has volume $\binom{n_w}{n_{wps}}$, so a conservative estimate of the maximum number of symbols that can exist is:

$$\frac{\binom{n_w}{n_{wps}}}{\sum_{i=0}^{n_{wps}(1-\frac{p_s}{m})} \binom{n_{wps}}{i}\binom{n_w-n_{wps}}{i}}$$

This yields a satisfactorily large capacity for symbols. For the experiments described above, with $n_w = 10000$, $n_{wps} = 100$, $p_s = 0.8$ and a maximum of 6 concurrent symbols, we find that the capacity is 1.167×10^{12} distinct symbols.

4.3 Performance Degradation

We expect that the performance of the algorithm will degrade gracefully as the channel capacity is reduced. As the average Hamming distance between symbols drops, the chance that a combination of other symbols will overlap to produce a spurious recognition or interfere with the inflection being transmitted rises. Since symbols receiving too much interference are discarded, the algorithm will tend to break up clusters of symbols and move toward an efficient filling of symbol space. Thus, reducing the ratio n_w/n_{wps} ought to cause the transmission errors to rise gradually and smoothly. In practice we find that this is in fact the case, as shown in Figure 4.3.

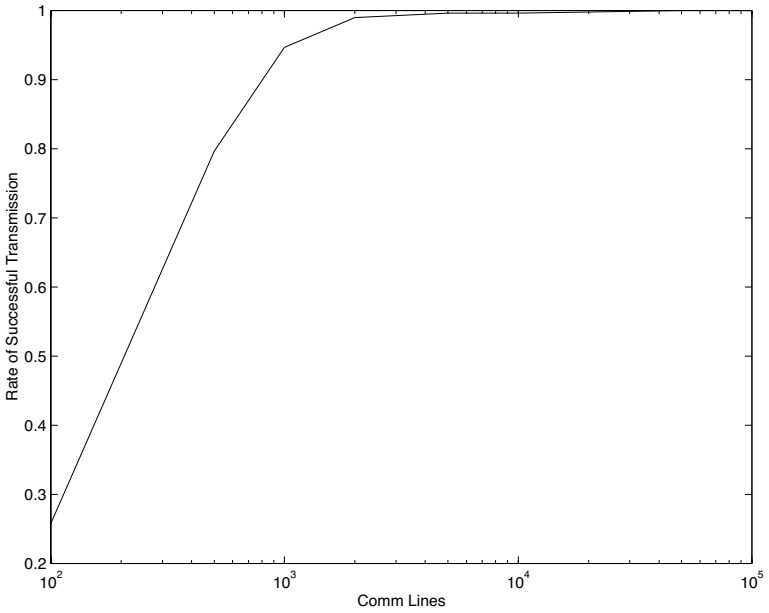
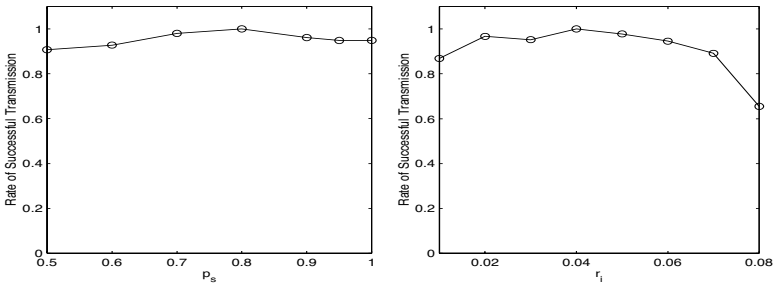


Figure 4.3: Number of comm lines versus transmission robustness. Horizontal axis is n_w from 100 to 100,000. Vertical axis shows the symbols and inflections correctly received per symbol and inflection transmitted (spurious receptions count against this as well) over the course of 200 test cycles on a system trained with 50 nouns, 20 verbs and 4 noun-roles, $n_{wps} = 20$, $p_s = 0.8$. Accuracy degrades smoothly with decreased channel capacity.



(a) Variation of p_s

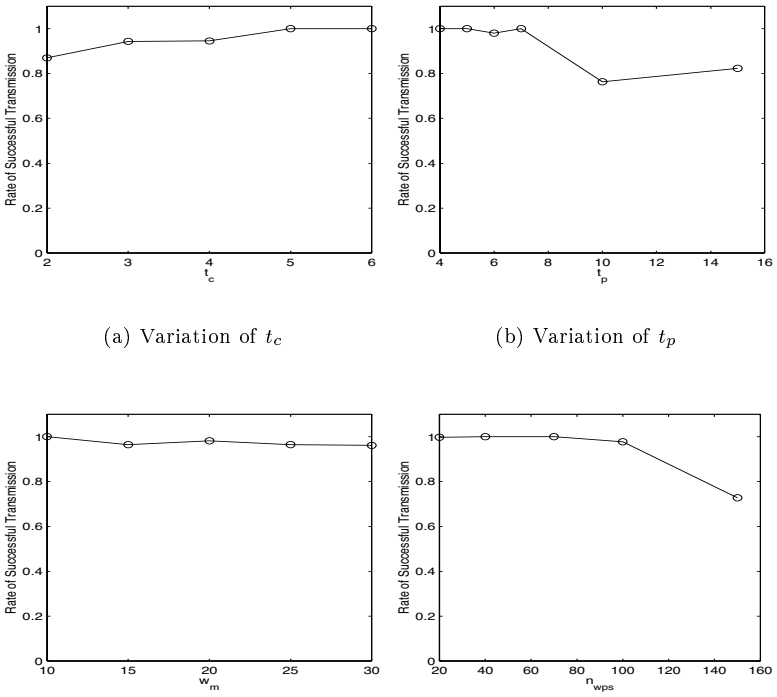
(b) Variation of r_i

Figure 4.4: Variation in performance as each parameter is varied. For each graph, the horizontal axis shows the value of the parameter being varied and the vertical axis shows the fraction of symbols and inflections correctly received per symbol and inflection transmitted. Measurements are the average values over the course of 10 runs of 200 test cycles, as in 4.3. For each run of test cycles, the systems were trained with 50 nouns, 20 verbs and 4 noun-roles, with base parameter values $p_s = 0.8$, $r_i = 0.05$, $t_c = 4$, $t_p = 6$, $w_m = 20$, $n_{wps} = 100$, and $n_w = 10000$. All parameters in the system can tolerate small variations without serious degradation in performance. This figure shows variation in p_s and r_i. Figure 4.5 shows variation in t_c, t_p, w_m, and n_{wps}

4.4 Parameter Variation

The values of the parameters used in the experiments above were not carefully chosen. Rather, I made a guess at a reasonable value for each parameter, expecting that the algorithm should not be very sensitive to the parameter values. (If it were, then I could hardly claim it was a robust algorithm!)

To test this, I ran a series of experiments in which I trained and tested the system with one of the parameters set to a different value. For each value for each parameter I ran 10 experiments: Figure 4.5 shows the performance of the algorithm as a function of parameter value for each of the six parameters p_s, r_i, t_c, t_p, w_m, and n_{wps}. (n_w is excluded because its variation is evaluated in the preceding section) As predicted, the performance of the algorithm is good over a wide range of values for each variable.



(a) Variation of t_c

(b) Variation of t_p

(c) Variation of w_m

(d) Variation of n_{wps}

Figure 4.5: Variation in performance as each parameter is varied, as in Figure 4.4, for t_c, t_p, w_m, and n_{wps}

4.5 Dissimilar Features

The operation of this algorithm is not confined to agents presented with identical features. If dissimilar but correlated feature sets are presented, then the similarities will be discovered and communicated in the language shared between the two agents. Features which are not shared will be transmitted as well, but will never be successfully interpreted by the listening agent.

I ran an experiment to confirm this, in which examples generated with 7 nouns, 6 verbs and 4 noun-roles were run through a filter before being input to the feature lines of each agent. The filter on the first agent added 0-2 random words from a set of 4, and split one noun into a pair of features. The filter on the second agent added precisely one random word from a set of four, remapped the verbs onto a set of overlapping component features, and relabelled one of the noun-roles. The system was then trained for 200 cycles with $n_w = 1000$ and $n_{wps} = 20$.

The final results showed that, as expected, the system had learned a shared vocabulary despite the handicaps imposed. The random words added to each side were ignored, since their appearance in the features of one agent was uncorrelated to the features seen by the other agent. The noun consistently split into a pair of features for the first agent was interpreted as a single symbol by the second agent, and communicated successfully in both directions. The relabelled noun-role in the second agent was also communicated unerringly. The verbs remapped onto component features for the second agent, however, were not able to be fully communicated. This was not unexpected, and occurred as predicted. The verb remapping was as follows:

$$move \rightarrow go$$

$$retreat \rightarrow go, from$$

$$approach \rightarrow go, to$$

$$touch \rightarrow go, contact$$

$$eat \rightarrow ingest, contact$$

$$fear \rightarrow fear$$

Since symbols are differentiated by intersection, the translation of several different symbols to the "go" symbol prevented "go" from being successfully communicated in either direction. Similarly, "contact" was impaired by its mapping to both "eat" and "touch". As a result,

"move" and "touch" could not be reliably communicated. All of the other verbs, however, having some unique component, were able to be communicated between the two agents successfully.

Chapter 5

Contributions

I have created a system which addresses the question of how peer components in a distributed intelligence system might construct a communication system. The design of the system is limited by restrictions inspired by real-world minds, with the hope that the good properties of real-world minds will prove more easy to develop in this restricted domain.

The algorithm which I have constructed allows two agents to create such a communications system. Moreover, they can do so rapidly, and with a cost linear in the size of the vocabulary. It operates robustly and degrades gracefully in adverse circumstances, and can even interface between related but dissimilar sets of observations.

There are some obvious possible applications of this research in the domains of networking, microchip design, and multi-agent robotics. In networking, communication protocols might be discovered or created automatically by agents which bootstrap based on shared information, and might prove resilient in the face of problems, given the ability to degrade gracefully. A significant problem in microchip design is the precision interconnects required to connect chips together: this has typically been addressed by having a few large wires between chips and accepting a much lower interchip bandwidth. A chip-to-chip communication system based on this research would not need precision interconnects, but could manufacture a communications system, possibly increasing bandwidth significantly. Finally, in the field of multi-agent robotics, this work could be used to help a group of robots communicate more reliably in the face of noise.

More importantly, however, this work provides a springboard for thinking about how to get about the practical business of building

intelligent systems from components which are truly peers. There is much work to be done in this area, and I believe that this thesis is a good first step.

5.1 Vision

If this thesis is a first step, however, it begs the questions, "What is the next step?" and "Where is this going?"

Artificial intelligence research has long been fettered by a tendency to view the problem of intelligence at either too high or too low a level. GOFAI techniques like Newell and Simon's logical reasoner, or Winograd's blocks world, or Marr's vision ideas, postulate intelligence as a unified and universal system, where one high-level system, with appropriate inputs, should be able to solve all problems. The low-level revolution of the 80s erred in the other directions, with Brooks demanding the abolition of all representational structure, Feldman and Ballard requiring a system to be constructed at a based level of neurons, and GA/GP advocates counselling defeat and advocating that we have computers build hairy low-level designs for us.

Considering these design as analogous to a computer: the GOFAI people would design the computer as a single UTM covering all activities: disk activity, memory reads and writes, caching, network communication, computation. The connectionists and embodied intelligence people, on the other hand, would take a computer to be its transistors, and counsel understanding on the level of electrical interactions between MOSFETs. We in computer science should have known better: computer science has long recognized the vital importance of abstraction and modularity in the design of large systems. Our approach to designing human intelligence should be no different — and recent neuroscience evidence suggests that this may indeed be correct, and that the brain may have many "organs" — internal subdivisions which specialize in one area.

This is not to say that this is the only approach to building intelligence, nor that human intelligence is the only workable model of intelligence. Indeed, given that our intelligence is the result of the arbitrary kludgey workings of evolution, it would be quite surprising if this was the case! At present, however, I believe that it is most profitable to build systems architected after inspiration from the human mind, for several reasons.

First, the human mind is the only example we have of an indisputably intelligent architecture, and we would be foolish to cast this

wealth of information aside. Moreover, attempting to duplicate the capabilities of humans gives us a yardstick against which we can measure our work and determine how well we are succeeding.

Second, biological systems, including humans and human intelligence, demonstrate a resiliency and versatility in dealing with adverse circumstances compared to which our engineering systems are frighteningly brittle and inflexible. Resiliency and versatility are fundamental tenets of "intelligent behavior", and so it seems likely that the human brain has architectural insights to teach us, and thus we would be wise to take inspiration from the human brain in designing our architectures.

Third, although the brain is an organ of staggering complexity, there are some informational limits on how complex it may be. Recent research suggests the human brain may be a highly organized structure with many specialized components, rather than some sort of vast disorganized "neural pudding". There is a hard informational limit on this structure however, in that it must be specified by our genetic code, which is only about 1 gigabyte, too small to encode the full structure of the brain in detail. One way of reconciling the structure of the brain with the poverty of instructions available for constructing it is to imagine the brain as composed of a large number of small components, where each component has a well-specified design drawn from a relatively small library of component types. Moreover, there are limits on the sorts of computation which we can expect it to perform, due to hardware limitations. Neurons, as hardware, are relatively slow and imprecise devices, but they are very cheap, and it's easy to throw a lot of them at a problem in parallel. Thus, we may expect that imitating local capabilities of the brain is a tractable engineering task.

Finally, there is a wealth of new functional data emerging from neuroscience about the functioning of the human brain. With the advent of fMRI, neuroscientists have been able to identify many apparent distinct modules within the brain, and are beginning to uncover data on how they relate to one another. With this new data, then, comes the practical hope that we will now have enough data on brain structure to make intelligent decisions in designing our artificial analogs.

Note that I am couching these guidelines in terms of "inspiration" rather than faithful duplication. As computer scientists, we ought not to be too concerned about the transistor-level characteristics of the hardware or wetware on which our algorithms are running. Rather, we ought to be concerned about the functionality of small modules — at the TTL component level, as it were.

That, then, is where I believe our research should focus for the next decade or two: on building the mid-level "TTL" components of

intelligence, and building them in frameworks inspired and restricted by human intelligence.

Appendix A

Scheme Code

A.1 Algorithm

```
;; Talker
;; Wires: only driven values are listed:
;; ( (line 1/-1/X) )
;; NOTE: Right now, all asserted feature lines are assumed to be positive

(declare (usual-integrations))

(define num-wires 10000) ;; 1000
(define num-wires-per-symbol 100) ;;20
(define min-wires-per-symbol 20) ;;5
(define percent-match 0.8)
(define unary-percent-match 0.05)

;; 12/6 Reformatting internal-map bundles as follows:
;; Instead of (symbol (commlines)) they now become
;; (symbol (certaincommlines) (uncertaincommlines) uncertainfactor)
;; When listening, both certain and uncertain comm lines are considered
;; When talking, only certain comm lines are used
;; An uncertain comm lines become certain after surviving n transmissions
(define certainty-threshold 4)
(define certain-pruning-threshold 6)

;; line abstractions
;; returns a line-bundle with the new value
;; On a conflict between values, the line becomes unasserted
;; This is represented by an "X" on the line
(define (assert-line line-bundle line value)
  (let* ((result (split line-bundle
                        (lambda (x) (eq? (first x) line))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (if pos
        (if (equal? (second (car pos)) value)
            line-bundle
            (cons (list line 'x) neg))
        (cons (list line value) line-bundle))))

;; this can put inflections on lines as well as simple values
```

```
(define (assert-feature-line line-bundle line inflection value)
  (let* ((result (split line-bundle
                       (lambda (x) (eq? (first x) line))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (if (and pos (or (not inflection)
                     (equal? (second (car pos)) inflection)))
        (if (or (not inflection) (equal? (third (car pos)) value))
            line-bundle
            (cons (list line inflection 'x) neg))
        (cons (list line inflection value) line-bundle))))

;; clean-bits: returns only the non-conflicted bits
(define (clean-bits line-bundle)
  (list-transform-negative line-bundle (lambda (x) (eq? (second x) 'x))))

;; returns the value found on the line, or zero if it's not asserted
;; Conlicted bits *are* returned as an "X"
;; Thus, there are four possible results: 1,-1,0,x
(define (test-line line-bundle line)
  (let ((result (list-search-positive line-bundle
                  (lambda (x) (eq? (first x) line)))))
    (if result (second result) 0)))

;; can check the inflection on feature-lines
(define (test-inflection line-bundle line)
  (let ((result (list-search-positive line-bundle
                  (lambda (x) (eq? (second x) line)))))
    (if result (second result) 0)))

;; listen & talk return (cons internal-map foo-line)
;; Two cases to handle:
;; 1. Features in internal-map
;;    If asserted on feature-lines: modify internal-map for conflicts
;;    else: if enough matches, assert on feature-lines
;; 2. Features asserted, but not in internal-map
;;    add new feature to internal-map
;; Returns (cons internal-map feature-lines
;; 1/2: added spokefirst
;; 1/4: changed "new features" to be features *heard* for the first time,
;;      regardless of whether they've been spoken before.
;;   NOTE: as it's set up, there's some overkill. Really, the critical
;;     factor is whether c-a is an empty set of not at symbol-creation.
;;     However, I'm keeping it this way since it's a bit more "pure"
(define (listen spokefirst comm-lines feature-lines internal-map pre-map comm-spoke)
  (let* ((smap (if internal-map (second internal-map) #f))
         (nmap (if internal-map (first internal-map) #f))
         (res (split smap
                (lambda (x)
                  (and (equal? 0 (test-line feature-lines (first x)))
                       (equal? 0 (test-inflection feature-lines (first x))))))))
         (stimfeat (second res))
         (smap-check (first res))
         (res2 (split stimfeat
                 (lambda (x)
                   (or spokefirst
                       (< 0 (find-symbol-certainty internal-map (first x))))))))
         (newfeat (second res2))
         (c-a (list-transform-negative comm-lines
                (lambda (x) (member x comm-spoke))))
         (newfeat
          (map (lambda (x)
```

```
               (list (car x) (find-symbol-codes internal-map (car x))
                     (map first c-a) 1))
             (second res2)))
       (smap-assert (if (not spokefirst)
                        (listen-resolve-conflicts spokefirst comm-lines
                                                  (first res2))
                        (first res2))))
    (list (list (if (not spokefirst)
                    (nsquelch (update-nmap nmap comm-lines
                                           feature-lines stimfeat))
                    nmap)
                (append smap-check smap-assert newfeat))
          (listen-assert-features comm-lines feature-lines nmap smap-check))))



;; returns a new internal-map with the codes changed/added
;; If a code-mapping drops below a minimum size, it is considered
;; worthless and discarded. This can happen to either new codes or
;; codes being changed.
;; 12/6: modified to handle certainty
(define (set-symbol-codes imap symbol codes)
  (let* ((smap (if imap (second imap) #f))
         (nmap (if imap (first imap) #f)))
    (list nmap (set-smap-codes smap symbol codes))))

(define (find-smap-codes smap symbol)
  (let* ((elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (second elt) #f)))

(define (find-smap-uncertains smap symbol)
  (let* ((elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (third elt) #f)))

(define (find-smap-certainty smap symbol)
  (let* ((elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (fourth elt) #f)))

(define (set-smap-codes smap symbol codes)
  (let* ((result (split smap
                        (lambda (x) (eq? (first x) symbol))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (if (< (+ (length (first codes)) (length (second codes)))
           min-wires-per-symbol)
        (begin
          neg)
        (if pos
            (cons (cons symbol codes) neg)
            (cons (cons symbol codes) smap)))))

;; sets the code for an inflection
(define (set-inflection-code imap inflection ucode)
  (let* ((smap (if imap (second imap) #f))
         (nmap (if imap (first imap) #f))
         (result (split nmap
                        (lambda (x) (eq? (first x) inflection))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (list (cons (list inflection ucode) neg) smap)))
```

```
;; The purpose of this function is to shrink the symbols in the
;; association DB in order to have more precise interpretations
;; of which lines represent a given symbol. (NOTE: explain more clearly)
;; Implemented by:
;; 12/6: adds 1 to the certainty of each symbol.
;; 1/2: added spokefirst
;; Certain lines are not filtered until the symbol has become certain
(define (listen-resolve-conflicts spokefirst comm-lines mappings)
  (let loop ((map-list mappings) (new-imap mappings))
    (if map-list
        (let* ((elt (first map-list))
               (ccodes (find-smap-codes new-imap (first elt)))
               (cnewcodes (list-transform-negative ccodes
                            (lambda (x)
                              (eq? 0 (test-line comm-lines x)))))
               (ucodes (find-smap-uncertains new-imap (first elt)))
               (unewcodes (list-transform-negative ucodes
                            (lambda (x)
                              (eq? 0 (test-line comm-lines x)))))
               (newcert (+ (if spokefirst 0 1)
                           (find-smap-certainty new-imap (first elt))))
               (newcodes
                (if (>= newcert certainty-threshold)
                    (if (>= newcert certain-pruning-threshold)
                        (list (->uniset (append cnewcodes unewcodes))
                              '() newcert)
                        (list (->uniset (append ccodes unewcodes))
                              '() newcert))
                    (list ccodes unewcodes newcert))))
          (loop (cdr map-list)
                (set-smap-codes new-imap (first elt) newcodes)))
        new-imap)))

;; This functions asserts a feature line if enough of its associated
;; comm-lines are lit up
;; 12/6: Only certain comm lines are considered.
(define (listen-assert-features comm-lines feature-lines nmap mappings)
  (fold-left (lambda (features imap-elt)
               (let* ((ll (second imap-elt)) ;; line-list
                      (fll (list-transform-negative ll
                             (lambda (x)
                               (eq? 0 (test-line comm-lines x)))))
                      (num-1s (length (list-transform-positive ll
                                        (lambda (x)
                                          (eq? -1 (test-line comm-lines x))))))
                      (num1s (length (list-transform-positive ll
                                       (lambda (x)
                                         (eq? 1 (test-line comm-lines x)))))))
                 (if (and ll (>= (/ (length fll) (length ll)) percent-match))
                     (let loop ((inflections
                                 (unary-match (/ num1s (+ num1s num-1s))
                                              nmap))
                                (f features))
                       (if inflections
                           (loop (cdr inflections)
                                 (assert-feature-line f (first imap-elt)
                                                      (car inflections) 1))
                           (assert-feature-line f (first imap-elt) #f 1)))
                     features)))
             feature-lines mappings))

(define (unary-match ucode mappings)
```

```
  (map first (list-transform-positive mappings
               (lambda (x) (< (abs (- (second x) ucode))
                              unary-percent-match)))))

;; given the nmap, adjusts interpretations
;; for features in the nmap
;;   if inflection on featurelines
;;      get uvalue of comm-lines
;;        if more than .5xu-%match away, jump to value
;;   else
;;      test if stimulated by any comm-lines
;;        if so, randomly reselect value
(define (update-nmap nmap comm-lines feature-lines stimfeat)
  (let loop ((tnmap nmap))
    (if
     (not tnmap)
     #f
     (let* ((infl (car tnmap))
            (res (map (lambda (x)
                        (let* ((lines (map (lambda (y)
                                             (test-line comm-lines y))
                                           (second x)))
                               (numbits (length lines))
                               (num1s (- numbits (length (delq 1 lines))))
                               (num-1s (- numbits (length (delq -1 lines))))
                               (numxs (- numbits (length (delq 'x lines)))))
                          (if (= numxs 0) ;; for now, only on clean ones
                              (if (>= (/ (+ num1s num-1s) numbits)
                                      percent-match)
                                  (list (first x) (/ num1s (+ num1s num-1s)))
                                  #f)
                              'invalid)))
                      stimfeat))
            (tmap (delq #f res))
            (ifeat (list-search-positive feature-lines
                     (lambda (x) (eq? (first infl) (second x))))))
       (if (member 'invalid tmap)
           (cons infl (loop (cdr tnmap)))
       (if ifeat
           (let* ((res (list-search-positive tmap
                         (lambda (x) (eq? (first ifeat) (first x)))))
                  ;; if no value in tmap, don't change the inflection value
                  (uvalue (if res (second res) (second infl)))
                  (dif (- uvalue (second infl))))
             (if (> dif (/ unary-percent-match 2))
                 (cons (list (first infl) uvalue) (loop (cdr tnmap)))
                 (cons infl (loop (cdr tnmap)))))
           (let ((res (list-search-positive tmap
                        (lambda (x) (< (abs (- (second x) (second infl)))
                                       unary-percent-match)))))
             (if res
                 (cons (list (first infl) (random 1.0))
                       (loop (cdr tnmap)))
                 (cons infl (loop (cdr tnmap)))))))))))

;; nmap = (list (list infl val))
;; nsquelch kills mappings which are too close together
(define (nsquelch nmap)
  (let loop ((m (random-permutation nmap)))
    (if m
        (let ((res (list-search-positive (cdr m)
                     (lambda (x) (< (abs (- (second x) (second (first m))))
```

```
                                (* 2 unary-percent-match))))))
          (if res
              (loop (cdr m))
              (cons (car m) (loop (cdr m)))))
        #f)))

;;  Returns the Comm-codes of a symbol in an internal representation
;;  12/6: Only returns the *certain* codes
(define (find-symbol-codes imap symbol)
  (let* ((smap (if imap (second imap) #f))
         (elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (second elt) #f)))

;; find-symbol-uncertains: returns list of *uncertain* codes
(define (find-symbol-uncertains imap symbol)
  (let* ((smap (if imap (second imap) #f))
         (elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (third elt) #f)))

;; find-symbol-certainty: returns the certainty value
(define (find-symbol-certainty imap symbol)
  (let* ((smap (if imap (second imap) #f))
         (elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (fourth elt) #f)))

;; find-inflection-code: returns the unary code for a given inflection
(define (find-inflection-code imap inflection)
  (let* ((nmap (if imap (first imap) #f))
         (elt (list-search-positive nmap
                (lambda (x) (eq? (first x) inflection)))))
    (if elt (second elt) #f)))

;; Assert-line on many different lines ((line value) (line value) ...)
(define (assert-line-multiple line-list linebundle)
  (if (null? line-list)
      linebundle
      (assert-line-multiple (cdr line-list) (assert-line linebundle
(caar line-list) (cadar line-list)))))

;; Generates Random Comm Codes
(define (random-comm-codes wps wires)
  (if (> wps 0)
      (cons (+ (random wires) 1) (random-comm-codes (- wps 1) wires))
      #f))

;; unary-signal: given a set of wires and a unary code, returns a list
;; of wires with appropriate bits on them. Signal is the %age of 1s.
(define (unary-signal wires ucode)
  (let* ((num1s (round->exact (* ucode (length wires))))
         (num-1s (- (length wires) num1s))
         (bits (random-permutation (append (make-list num1s 1)
                                           (make-list num-1s -1)))))
    (map list wires bits)))

;; Main Talk Procedure
;; 12/6: only outputs certain symbols; all newly generated randoms are certain
;; Internal-Maps is now a list of two elements. The first is inflection maps,
;; the second is symbol maps
(define (talk comm-lines feature-lines internal-maps)
  (define (loop feature-lines)
    (if (null? feature-lines)
        #f
```

```
        (let* ((cur-elmt (car feature-lines))
               (symbol (first cur-elmt))
               (inflection (second cur-elmt))
               (comm-codes (find-symbol-codes internal-maps symbol))
               (infl-code (find-inflection-code internal-maps inflection)))
          (cond ((null? comm-codes)
                 (set! internal-maps
                       (set-symbol-codes
                        internal-maps symbol
                        (list (random-comm-codes num-wires-per-symbol
                                                 num-wires) '() 0)))
                 (loop feature-lines))
                ((null? infl-code)
                 (set! internal-maps
                       (set-inflection-code internal-maps inflection
                                            (random 1.0)))
                 (loop feature-lines))
                (else
                 (append (unary-signal comm-codes infl-code)
                         (loop (cdr feature-lines)))))))))

  (if (null? feature-lines)
      (list internal-maps comm-lines)
      (let* ((newlines (assert-line-multiple (loop feature-lines) comm-lines))
             (newmaps  internal-maps))
        (list newmaps newlines))))
```

A.2 Testbed

```
;; runs simple tests through:

(declare (usual-integrations))

(cf "utils.scm")
(cf "talk2.scm")
(cf "talker.scm")
(cf "runtest.scm")

(load "utils")
(load "talk2")
(load "talker")
(load "runtest")

(define (make-log name)
  (let ((fout (open-output-file name)))
    (lambda (x) (if (eq? x 'flush-buffer)
                    (flush-output fout)
                    (pp x fout)))))
;; exa of use:
;; (define thelog (make-log "thefile.out"))
;; (thelog '(this is a list I want to log))
;; To flush to disk:
;; (thelog 'flush-buffer)


;; The system consists of two talkers, each with a feature-bus coming
;; in, and a comm-line connecting.
;; The state of this system can be described as a list of five elements:
;; (feat1 talker1 comm talker2 feat2)
;; Starting pattern for a instance
;; ( ((f1 1) (f2 2)) imap1 () imap2 () )
```

```
;;       or
;; ( ((f1 1) (f2 2)) imap1 () imap2 ((f1 1) (f2 2)) )
;; Run a test by:
;;   update imap1, update imap2: check for quiescence


;; A feature set is defined as follows:
;; There are five roles which can be expressed:
;; subject, object, action, instrument, place
;; Actions come from category "verb" and anywhere from 0-2 can be asserted
;; All others are nouns and 2-4 can be asserted
;; (YES, IT'S PRETTY ARBITRARY: I JUST WANT A DECENT-SIZED SPACE)
;; How big is it? 28 nouns, 6 verbs. avg 4 words/sentence.

; Normal Set
;(define nouns '(bob jim mary icepick shovel table lab))
;(define verbs '(move approach retreat touch eat fear))
;(define noun-roles '(subject object instrument place))

; Mega Set - 50 nouns, 20 verbs
(define nouns '(bob jim mary icepick shovel table lab fred bill classroom
                leg cup butterfly dog cat turtle door window car hammer
                keyboard coffee danish pencil pen eraser wall socket book ed
                wheel chainsaw gun kite bedroom shower beach shoe light dark
                hat office house apple banana flea vampire stapler kim joe))
(define verbs '(move approach retreat touch eat fear zap feel fly throw
                catch push hit stab tickle hurt love hate want ignite))
(define noun-roles '(subject object instrument place))


(define (generate-features)
  (let loop ((nnoun (+ 2 (random 3)))
             (nverb (random 3))
             (features '())
             (roles '()))
    (let ((vp (list-transform-negative verbs (lambda (x) (memq x features))))
          (np (list-transform-negative nouns (lambda (x) (memq x features))))
          (nr (list-transform-negative noun-roles (lambda (x) (memq x roles)))))
      (if (<= nnoun 0)
          (if (<= nverb 0)
              (map (lambda (x y) (if y
                                     (list x y 1)
                                     (list x 'verb 1)))
                   features roles)
              (loop 0 (- nverb 1)
                    (cons (list-ref vp (random (length vp))) features)
                    (cons #f roles)))
          (loop (- nnoun 1) nverb
                (cons (list-ref np (random (length np))) features)
                (cons (list-ref nr (random (length nr))) roles))))))


(define cycle-log (make-log "cycle.log"))

(define (run-cycles initial-state i terminate)
  (let* ((features (generate-features))
         (istest (< 0.8 (random 1.0)))
         (testwhich (< 0.5 (random 1.0)))
         (state (list (if istest (if testwhich features #f) features)
                      (second initial-state)
                      #f
                      (fourth initial-state)
```

```
                    (if istest (if testwhich #f features) features))))
    (let ((newstate (iterate state)))
      (if istest
          (if testwhich
              (pp (list i 'test2 (first state) 'returns (fifth newstate)))
              (pp (list i 'test1 (fifth state) 'returns (first newstate))))
          (pp (list i 'train (first state))))
      (cycle-log (list 'step i features state newstate))
      (cycle-log 'flush-buffer)
      (if (or (eq? terminate -1) (>= i terminate))
          newstate
          (run-cycles newstate (+ i 1) terminate)))))

(define cyclestate (run-cycles '(#f #f #f #f #f) 1 1000)) ;; 300

;; test a percent-correct measure
(let loop ((i 0) (successes 0) (state cyclestate))
  (let* ((features (generate-features))
         (state (list (if (< i 100) features #f)
                      (second state)
                      #f
                      (fourth state)
                      (if (< i 100) #f features))))
    (if (>= i 200)
        (pp `(Final results: ,successes sucesses out of ,i trials))
        (let ((newstate (iterate state)))
          (let ((fout (if (< i 100) (fifth newstate) (first newstate))))
            (if (and (equal-set? features (fifth newstate) equal?)
                     (equal-set? features (first newstate) equal?))
                (begin
                  (cycle-log (list 'test i features 'succeed))
                  (cycle-log 'flush-buffer)
                  (loop (+ i 1) (+ successes 1) newstate))
                (begin
                  (cycle-log (list 'test i features fout 'fail))
                  (cycle-log 'flush-buffer)
                  (loop (+ i 1) successes newstate)))))))))

(declare (usual-integrations))

(define (make-log name)
  (let ((fout (open-output-file name)))
    (lambda (x) (if (eq? x 'flush-buffer)
                    (flush-output fout)
                    (pp x fout)))))


(define get-feat1 first)
(define get-imap1 second)
(define get-comm third)
(define get-imap2 fourth)
(define get-feat2 fifth)

;; talk returns: (imap comm)
;; prune-conflicts returns: imap
;; listen returns: (imap feat)
;; iterate operates in the following manner:
;; 1. comm lines start blank.
;; 2. have everything speak
;; 3. have everything resolve speaking conflicts
;; 4. have everything listen
```

```
;; Unfortunately, if we do this in perfect order, then the synchrony
;; makes it difficult to separate the outgoing and incoming transmissions,
;; most particularly, to determine which bits of a feature are correct.
;; We resolve this by ordering the transmissions such that one of the
;; two hears the others transmissions before its transmission clutters
;; the wires. (If, of course, only one is transmitting, then it cannot
;; be pre-empted)

(define (iterate state)
  (let* ((tres1 (talk '() (get-feat1 state) (get-imap1 state)))
         (tres2 (talk '() (get-feat2 state) (get-imap2 state)))
         (comms (combine-comms (second tres1) (second tres2)))
         (whofirst (if (and (second tres1) (second tres2))
                       (if (eq? (random 2) 0) #t #f)
                       (if (second tres1) #t #f)))
         (lres1 (listen whofirst (if whofirst comms (second tres2))
                        (get-feat1 state) (first tres1)
                        (get-imap1 state) (second tres1)))
         (lres2 (listen (not whofirst) (if whofirst (second tres1) comms)
                        (get-feat2 state) (first tres2)
                        (get-imap2 state) (second tres2)))
         (newstate (list (second lres1) (first lres1) comms (first lres2)
                         (second lres2))))
    newstate)) ; for now, just doing a single learning pass

;; this merges a set of comm lines, turning transmissions into noise
(define (combine-comms . inputs)
  (fold-left
   (lambda (x y)
     (let* ((res1 (split x
                         (lambda (a)
                           (list-search-positive y
                             (lambda (b) (equal? (first b) (first a)))))))
            (res2 (split y
                         (lambda (a)
                           (list-search-positive x
                             (lambda (b) (equal? (first b) (first a)))))))
            (merged (map (lambda (a)
                           (let* ((v2 (second
                                       (list-search-positive (first res2)
                                         (lambda (b)
                                           (equal? (first b) (first a)))))))
                             (list (first a) (if (eq? v2 (second a))
                                                 (second a)
                                                 'x))))
                         (first res1))))
       (append (second res1) (second res2) merged)))
   '()
   inputs))
```

Bibliography

[1] Hearn, Robert. "Building Grounded Abstractions for Artificial Intelligence Programming." S.M. Thesis, Massachusetts Institute of Technology, June 2001

[2] Kirby, Simon. "Language evolution without natural selection: From vocabulary to syntax in a population of learners." Edinburgh Occasional Paper in Linguistics EOPL-98-1, 1998. University of Edinburgh Department of Linguistics.

[3] Kirby, Simon. "Learning, Bottlenecks and the Evolution of Recursive Syntax." Linguistic Evolution through Language Acquisition: Formal and Computational Models edited by Ted Briscoe. Cambridge University Press, in Press.

[4] Lynch, Nancy. Asynchronous Systems Model. In *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc, San Francisco, California, 1996. Chapter 8, pages 199-234.

[5] Minky, Marvin. "K-Lines: A Theory of Memory" MIT AI Lab Memo 516, June 1979.

[6] Minsky, Marvin. *The Society of Mind*. Simon & Schuster, Inc, New York, 1985.

[7] Mooers, Calvin. "Putting Probability to Work in Coding Punched Cards: Zatocoding (Zator Technical Bulletin No. 10), 1947. Reprinted as Zator Technical Bulletin No. 65 (1951).

[8] Roy, Deb. "Learning from Sights and Sounds: A Computational Model" Ph.D. Thesis, MIT Media Laboratory, 1999.

[9] Yanco, Holly. "Robot communication: issues and implementations." Sc.M. Thesis, Massachusetts Institute of Technology, May, 1994

[10] Yip, Kenneth and Sussman, Gerald Jay. "Sparse Representations for Fast, One-Shot Learning." MIT AI Lab Memo 1633, May 1998.