THE TENTH WHITE HOUSE PAPERS
Graduate Research in Cognitive and Computing Sciences at Sussex

Editors:
John C. Halloran & Fabrice P. Retkowsky

CSRP 478

January 1998

ISSN 1350-3162

UNIVERSITY OF

SUSSEX
AT BRIGHTON

Cognitive Science
Research Papers

THE TENTH WHITE HOUSE PAPERS

Graduate Research in Cognitive and Computing Sciences at Sussex

CSRP 478

Editors:
John C. Halloran & Fabrice P. Retkowsky

January 1998

Contents

Dedication

The editors would like to dedicate the Tenth White House Papers to Sarah Parsowith for all the ‘extras’ she did while at COGS, such as editing the 1996 White House Papers and organising the Isle of Thorns workshop last year. Sarah left COGS a few months ago to go to Australia.

Preface

At the Isle of Thorns (a small village situated not far from Haywards Heath), stand a few white buildings. These buildings, which sometimes act as Sussex University's conference centre, are most of the time used as a playground by rabbits. However, every year, they are disturbed by a congregation of COGS research students. In accordance with this time-honoured tradition, 1998 will see the 11th Isle of Thorns workshop, where COGS students will gather to present their work, share some ideas, and spend the rest of the time socialising.

The White House Papers are an introduction to this year's workshop. They include papers written by PhD students in 1997, and shed some light on the diversity of the subjects studied at COGS. They include cognitive psychology, intelligent tutoring systems, logic, medical diagnosis, natural language processing, neural networks, robotics, software design, debugging and reuse.

The editors would like to thank all the PhD students who contributed to these White House Papers, as well as Matthew Henessy and the COGS Graduate Research Centre for funding the IoT workshop.

John Halloran
Fabrice Retkowsky
January 1998

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

An Extension of Defeasible Prolog

Ahti Pietarinen
ahtivp@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract Defeasible Prolog is a Prolog metainterpreter designed by Nute to implement nonmonotonic inference based on a defeasible logic. In this paper it is first shown how to give proof conditions for even-if conditions of Nute's defeasible logic that allow also the preemption of defeaters. These conditions are then implemented to the defeasible Prolog. Finally, some computational results are presented for the given examples.

1 Introduction

In Nute's defeasible reasoning (Nute, 1992, 1994a, 1994b), the main target is to give a formalised account of nonmonotonic defeasible inferences like "typically, φ's are ψ", "normally, φ's are ψ" or "usually, φ's are ψ". These inferences hold, if a defeasible theory does not contain other rules that represent contrary information leading to the conclusion that the state of affairs exemplified by former defeasible rules might not hold any more. In the presence of such rules, former inferences may become defeated, and hence defeasible reasoning steps into the nonmonotonic area: a set of conclusions does not always grow monotonically when new premises are added.

Defeasible Prolog (d-Prolog) is a Prolog metainterpreter designed by Nute (Nute, 1997) to implement some basic forms of nonmonotonic defeasible inference. To the ordinary syntax of Prolog new two-place operators "`:=`" and "`:^`" are added to represent a *defeasible implication* (ψ `:=` Φ, read as "typically, Φ's are ψ") and a *defeater* (ψ `:^` Φ, read as "if Φ, it can be taken as a reason to doubt that not ψ" or "if Φ, it might be that ψ"), respectively. Although it is not immediately clear what the intuitive difference between two rules is, their uses in the proof derivations are formalised slightly differently. In particular, a defeater does not give evidence to support some conclusion, for its intented meaning is just to interfere with the derivations from the defeasible rules. Ordinary Prolog implication "`:-`" has, however, its usual meaning. In d-Prolog, rule heads are literals (atomic sentences or their negations), and bodies are conjunctions of atomic sentences. Unlike in ordinary Prolog, literals, and hence also the heads of the rules may be explicitly negated with a one-place predicate "`neg`". As we shall see, the approach Nute advocates is entirely proof theoretic.

Following the formulation of Nute, a *defeasible theory* $\mathcal{T}$ is a tuple $\langle K, R\rangle$, where K is a finite set of literals and R is a finite set of rules. A proof of a literal ψ is defined as a proof tree t, where ψ is a root node of t and t is a finite labelled tree such that for every node n of t there is a theory $\mathcal{T}$ and some literal which is marked as positive (φ^+) or negative (φ^-). When the node contains φ^+, we know that the literal is *defeasibly derivable* from $\mathcal{T}$ with respect to the set of proof conditions Σ ($\mathcal{T} \vdash_\Sigma \psi$); on the other hand, if the node has the label ψ^-, it indicates that the literal is *demonstrably not derivable* from the theory $\mathcal{T}$ by the conditions in Σ ($\mathcal{T} \nvdash_\Sigma \psi$). Defeasible logic is defined as a set of conditions Σ on every

node of the tree t for which it is never the case that $\mathcal{T} \vdash_\Sigma \psi$ and $\mathcal{T} \nvdash_\Sigma \psi$. To solve the problem of which defeasible rules should be used to defeat other defeasible rules, Nute provides an explicit partial order "$\sqsupset$" between defeasible rules which is termed as a superiority relation. Thus, if a rule r_1 is *superior* to a rule r_2 ($r_1 \sqsupset r_2$), then it may be used to defeat the *inferior* rule r_2. In some cases, it is also possible to extract information about the partial order by using the method of more specific antecedent (*cf. e.g.* (Nute, 1992)). Defining superiority by antecedent specificity, the antecedent conditions of one rule are derivable from the antecedent conditions of another rule. However, an extra complication is needed, for some parts of the derivations in a proof tree may be based on just the subtheory of $\mathcal{T}$. Therefore, some labellings may vary on their admissible set of literals and rules upon which the proofs of literals in rules defined by more specific antecedents depend.

Different rules interact with each other in the following ways. The rule ψ `:=` Φ is *defeated* by the rule η `:=` X or η `:-` X, if the head η is either a negation of ψ or some literal which is explicitly stated as incompatible with ψ. Such a literal η is called *contrary* to ψ. Next, the rule ψ `:=` Φ is *undercut* by the defeater η `:^` X, if η is contrary to ψ and ψ `:=` Φ is not superior to η `:^` X. Finally, a defeasible rule ψ `:=` Φ or a defeater ψ `:^` Φ may also be *preempted*, whenever there exists a literal η contrary to ψ which is derivable from the theory, or there is an ordinary Prolog rule η `:-` X with a contrary head, or for the defeasible rule ψ `:=` Φ there is a rule η `:=` X, whose head η is contrary to ψ and which is superior to it, that is, if (η `:=` X) $\sqsupset$ (ψ `:=` Φ). Furthermore, the literals contained in the bodies X in any of the former rules must always be defeasibly derivable from the defeasible theory $\mathcal{T}$. The role of the preemption is to relax the assumption that applicable defeasible rules to defeat competing contrary rules must always be superior to every other competing rule. When the preemption is enabled, once the defeasible rule is defeated it loses its capacity to defeat any other rule.

We take the latest d-Prolog version (Nute, 1997) as our starting point and show how to extend it with the so called even-if rules and even-if proof conditions given to those rules. Even-if rules and even-if conditions without preemption were given in (Nute, 1994a), but they have not been implemented in d-Prolog. Our task is, thus, to first formulate these conditions in theoretical level such that they also enable preemption, and then show how they can be implemented in d-Prolog. Finally, we shall derive some computational results by computing some sizes of the proof trees for the given examples. Some of the corresponding new d-Prolog predicates are presented in the appendix.

2 New Rules for the Even-If Conditions

In addition to the basic defeasible reasoning, Nute has introduced even-if rules and conditions in the defeasible logic in (Nute, 1994a). Their target is to extend basic defeasible reasoning allowing new kinds of inference. Formally, an even-if rule is a rule ψ `:=` Ξ `|` Φ, and is read "if Φ, then typically ψ even if Ξ holds". Let us call the condition Φ the main condition and the condition Ξ the even-if condition. These rules are always defeasible, that is, they do not apply to the ordinary Prolog rules. Moreover, they allow a superiority relation to be defined by the method of more specific antecedent. The conditions for proper use of these rules have not, however, previously been given in a form that also enables the preemption of defeaters. Before formulating these conditions, to illustrate the use and applicability of even-if rules, consider following two examples.

Example 2.1 A typical Finn is envious (`envious(X) := finnish(X)`). A typical Lappman usually is not, however, envious even if he is a Finn (`neg envious(X) := finnish(X) | lappman(X)`). Tomi is a Lappman (`lappman(tomi)`). Clearly he must not be envious, whether a Finn or not. In both cases it can, indeed, be concluded tentatively that `neg envious(tomi)` is a derivation from the theory $\mathcal{T}$.

Example 2.2 Let us add to the previous theory a defeater, according to which a Lappman, who has not received EU funds may very well be envious (`((envious(X) :^ (lappman(X), neg funds(X)))`). If Tomi is not given funds (`neg funds(tomi)`), it is hard to say, with this amount of information, whether Tomi is envious or not. Perhaps the best thing is to refuse drawing any conclusion. In our extension of d-Prolog, both `neg envious(tomi)` and `envious(tomi)` are demonstrably not derivable from the $\mathcal{T}$, since the rule (`envious(X) :^ (lappman(X), neg funds(X))`) is not an acceptable rule to preempt an inferior rule (`neg envious(X) := finnish(X) | lappman(X)`), because preemption is only possible if the rule is defeated with a superior rule.

The conditions that are responsible for the correct even-if derivations are rather simple modifications to the conditions given in (Nute, 1994a) in the following respects. First, a substitution is removed for simplicity. Second, notation is slightly changed, since we deal with the fixed d-Prolog syntax. Third, the preemption of defeaters is taken into account by adding extra constraints for it.

Definition 2.1 *Let n be an arbitrary node of the proof tree t, K the set of literals, R the set of rules, and let $@\psi$ denote when ψ is a tentative conclusion derived from the theory $\mathcal{T} = \langle K, T\rangle$. The following conditions P^+ and P^- are defined for all the nodes n, the former for the literals that are defeasibly derivable and the latter for the literals that are demonstrably not derivable. We also require the main conditions to be nonempty.*

P^+: *The node n is labelled $\langle K, R, @\psi^+\rangle$, if either n has a child labelled $\langle K, R, \mathtt{neg}\ \psi^-\rangle$ or there exists a rule ψ `:=` Ξ `|` $\Phi \in R$ such that all of the following (1), (2) and (3) hold:*

- *(1) for every $\varphi \in \Phi$ a node n has a child labelled $\langle K, R, @\varphi^+\rangle$*
  (every literal in the main body is derivable)
- *(2) for every* `neg` ψ `:=` $H \in R$ *there exists $\eta \in H$ and a child of n labelled $\langle K, R, @\eta^-\rangle$*
  (every contrary rule must have a nonderivable literal in the body)
- *(3) for every* `neg` ψ `:=` Δ `|` $\Theta \in R$ *or* `neg` ψ `:^` $\Theta \in R$ *either (a) or (b) holds:*
  - *(a) there exists $\theta \in \Theta$ and a child of n labelled $\langle K, R, @\theta^-\rangle$*
    (every contrary rule must have a nonderivable literal in the main body)
  - *(b) all of the following (i), (ii) and (iii) hold:*
    - *(i) there exists $\varphi \in \Phi \cup \Xi$ and a child of n labelled $\langle \Theta, R, @\varphi^-\rangle$*
      (specificity: some literal in the antecedent conditions of the main rule is not derivable from the antecedent conditions of the contrary rules)
    - *(ii) for every $\theta \in \Theta$ there exists a child of n labelled $\langle (\Phi \cup \Xi), R, @\theta\varphi^+\rangle$*
      (specificity: every literal in the main condition of the contrary rule is derivable from the antecedent conditions of the main rule)
    - *(iii) for every $\varphi \in \Phi \cup \Xi$ there exists a child of n labelled $\langle K, R, @\varphi^+\rangle$*
      (preemption: every literal in the body of the main rule is defeasibly derivable).

P^-: *The node n is labelled $\langle K, R, @\psi^-\rangle$, if either n has a child labelled $\langle K, R, @\psi^+\rangle$ or both (1) and (2) hold:*

- *(1) for every ψ `:=` $\Phi \in R$, either (a) or (b) holds:*
  - *(a) there exists $\varphi \in \Phi$ and a child of n labelled $\langle K, R, @\varphi^-\rangle$*
    (a literal in the body is demonstrably not derivable)

(b) *there exists* ψ `:=` $H \in R$ *such that for every* $\eta \in H$ *a node* n *has a child labelled* $\langle K, R, @\eta^+ \rangle$
(some other rule with the same head has a derivable body)

(2) *for every* ψ `:=` Ξ `|` $\Phi \in R$, *either (a), (b) or (c) holds:*

(a) *there exists* $\varphi \in \Phi$ *and a child of* n *labelled* $\langle K, R, @\varphi^- \rangle$
(a literal in the head is demonstrably not derivable)

(b) *there exists* `neg` ψ `:=` $X \in R$ *such that for every* $\chi \in X$ *a node* n *has a child labelled* $\langle K, R, @\chi^+ \rangle$
(some contrary rule has a derivable body)

(c) *there exists* `neg` ψ `:=` Δ `|` $\Theta \in R$ *or* `neg` ψ `:^` $\Theta \in R$ *such that for every* $\theta \in \Theta$ *a node* n *has a child labelled* $\langle K, R, @\theta^+ \rangle$, *and either (i), (ii) or (iii) holds:*

(i) *there exists* $\theta \in \Theta$ *and a child of* n *labelled* $\langle (\Phi \cup \Xi), R, @\theta^+ \rangle$
(specificity: some literal in the main body of the rule is derivable from the body of the original rule)

(ii) *for every* $\varphi \in \Phi \cup \Xi$ *a node* n *has a child labelled* $\langle \Xi, R, @\varphi^+ \rangle$
(specificity: every literal in the body of the original rule is is derivable from the main body of the contrary rule)

(iii) *there exists* $\varphi \in \Psi \cup \Xi$ *and a child of* n *labelled* $\langle K, R, @\theta^- \rangle$.
(preemption: some literal in the body of the original rule is demonstrably not derivable).

Proposition 2.1 $\mathbf{P} = \mathbf{M} \cup \{SS^+, P^+, P^-\}$ *is a defeasible logic.* [1]

PROOF. On the depth of the proof tree t it can be shown that there is no theory $\mathcal{T}$ and a literal ψ such that $\mathcal{T} \vdash_{\mathbf{P}} \psi$ and $\mathcal{T} \nvdash_{\mathbf{P}} \psi$. □

3 Extending d-Prolog

To incorporate previous conditions to the latest version of d-Prolog (Nute, 1997), we add to its syntax a new two-place relation "`|`", which distinguishes between the primary condition Φ in the rule ψ `:=` Ξ `|` Φ and secondary even-if condition Ξ, as in the previous conditions P^+ and P^-.[2] We need to consider the following modifications and additions to the defeating and undercutting conditions that were given in (Nute, 1997).

(1) The rule ψ `:=` Ξ or ψ `:-` Ξ is defeated (as implemented in the predicate `defeated/2`), if there exists an even-if rule φ `:=` Ξ `|` Φ, whose head φ is contrary to ψ, whose body Φ without the even-if condition Ξ is defeasible derivable, and whose even-if condition Ξ is a body of such a rule r which is not superior to the rule φ `:=` Ξ `|` Φ.

(2) The rule ψ `:=` Ξ `|` Φ is defeated, (i) if there exists a rule φ `:=` Θ whose head φ is contrary to ψ, Θ is defeasibly derivable, and the condition Ξ is not the head of any defeasible rule, which is not superior to the rule φ `:=` Θ, or (ii) if there exists a rule φ `:=` Φ `|` Θ, whose head φ is contrary to ψ, Θ is defeasibly derivable, and the even-if condition Φ is the head of a defeasible rule r which is not superior to the rule φ `:=` Φ `|` Θ.

[1] $\mathbf{M}$ is a monotonic core of a defeasible logic consisting of the four basic conditions, and $\mathbf{SS}^+$ is a semi-strictness condition. They are both defined, for example, in (Nute, 1992). With the semi-strictness condition, an ordinary rule ψ `:-` Φ may defeat another ordinary rule `neg` ψ `:-` Ψ if the antecedent Ψ is only defeasibly derivable.

[2] In the d-Prolog predicates given in the appendix the symbol "`|`" is replaced with the symbol "`#`".

(3) The rule ψ := Ξ | Θ is undercutted (`undercut/2`), if there exists a defeater φ :^ Θ, whose head is contrary to ψ, Θ is defeasibly derivable, and ψ := Ξ | Θ is not superior to it.

The preemption of defeaters (`preempted/2`) is almost analogical to the defeating conditions (see Appendix A).

There are also two new definitions for the defeasible derivability (`def_der/2`), three definitions for defining superiority relation with the defeasible specificity (`sup_rule/2`) and some other definitions for syntax and occurrences of the even-if rules. Some of these predicates are described in the appendix, and we omit the details here.

4 Computational Results

To get a glimpse of the (in)efficiency of the defeasible inferences in d-Prolog, I have executed several runs for many typical benchmark problems for nonmonotonic defeasible reasoning. Here I show the results for the examples 2.1 and 2.2 executed in the extended d-Prolog; those results are summarised in the Table 1. Note how enabling the preemption of the defeaters greatly increases the sizes of the proof trees. In general, the sizes are clearly enormous, and there are not, therefore, reasonable grounds to expect defeasible reasoning of this kind to be tractable.

	Example 2.1	
preempt	`@ envious(tomi)`	`@ neg envious(tomi)`
yes	23	90
no	53	98
	Example 2.2	
preempt	`@ envious(tomi)`	`@ neg envious(tomi)`
yes	23	138
no	53	270

Table 1: Sizes of the proof trees for the queries '`@ envious(tomi)`' and '`@ neg envious(tomi)`' for the predicates presented in the examples 2.1 and 2.2.

5 Conclusion

There are, however, straightforward strategies to limit the complexity of defeasible reasoning. For example, the sizes of the proof trees can be restricted to simplify inferences in the following two ways:

(1) Restrict the depth of the proof tree by generating paths only up to the certain fixed limit. After reaching the limit, the proof backtracks to the earlier nodes.

(2) Restrict the branching factor of the proof tree by keeping the size of the rule set *R* small and also the maximum number of conjunctions in every rule reasonably small.

The former method, however, has a notorious side effect: since the nature of the "argumentative" defeasible inference is depth-first search (some labels may depend on the other distant labels), constraining it may cause horizonal effects. Moreover, limiting maximal depth amounts to the cheap method of loop checking. I propose that better method, when there is a fear of cyclic dependency graphs, would be to stratificate the predicates so that they do not depend on each other in a circular manner. This is an obvious future step that ought to be done if one wants to improve the performance of d-Prolog.

The latter method is hence more appropriate, since it reflects the nature of defeasible reasoning better. In argumentation only the most pertinent and plausible arguments should be used to support arguments.

In conversations, it is often even necessary to lay size restrictions to the allowable set of one's assertions. Usually, even one or two defeaters, "objections", are enough to render opposite views implausible.

A An Extension of d-Prolog

```
% only a small fragment of the extension of d-Prolog is presented here %

init :- op(1100,fx,@),
        op(900,fx,neg),
        op(1100,xfy,:=),
        op(1100,xfy,:^),
        op(1100,xfy,#).

:- dynamic((neg)/1, (:=)/2, (:^)/2, (#)/2).
:- multifile((neg)/1, (:=)/2, (:^)/2, (#)/2).

def_der(KB,Goal) :-
        preemption,
        def_rule(KB,(Goal := (ConditionX # Condition))),
        \+ (contrary(Goal,Contrary1),
            strict_der(KB,Contrary1)),
            def_der(KB,Condition),
        \+ (contrary(Goal,Contrary2),
            clause(Contrary2,Condition2),
            Condition2 \== true,
            def_der(KB,Condition2)),
        \+ (contrary(Goal,Contrary3),
            def_rule(KB,(Contrary3 := (Condition4 # Condition3))),
            def_der(KB,Condition3),
            \+ (preempted(KB,(Contrary3 := (Condition4 # Condition3))))),
        \+ (contrary(Goal,Contrary5),
            def_rule(KB,(Contrary5 := Condition5)),
            def_der(KB,Condition5),
            \+ (preempted(KB,(Contrary5 := Condition5)))),
        \+ (contrary(Goal,Contrary6),
            (Contrary6 :^ Condition6),
            def_der(KB,Condition6),
            \+ (preempted(KB,(Contrary6 :^ Condition6)))).

defeated(KB,(Head := (Body1 # Body))) :-
        contrary(Head,Contrary),
        def_rule(KB,(Contrary := (Condition2 # Condition))),
        def_der(KB,Condition),
        \+ sup_rule((Head := (Body1 # Body)),
                    (Contrary := (Condition2 # Condition))),!.

undercut(KB,(Head := (Body1 # Body))) :-
        contrary(Head,Contrary),
        (Contrary :^ Condition),
        def_der(KB,Condition),
        \+ sup_rule((Head := (Body1 # Body)),(Contrary :^ Body)),!.

preempted(KB,(Head := (Body1 # Body))) :-
        contrary(Head,Contrary),
        def_rule(KB,(Contrary := (Condition2 # Condition))),
        def_der(KB,Condition),
        sup_rule((Contrary := (Condition2 # Condition)),
```

```
(Head := (Body1 # Body))),!.
```

References

Nute, D. (1992). Basic defeasible logic. In Fariñas del Cerro, L., & Penttonen, M. (Eds.), *Intensional Logics for Logic Programming*, pp. 125–154. Oxford University Press, Oxford.

Nute, D. (1994a). A decidable quantified defeasible logic. In Prawitz, D. Skyrms, B., & Westerståhl, D. (Eds.), *Logic, Methodology and Philosophy of Science IX*, pp. 263–284. Elsevier Science B.V., Holland.

Nute, D. (1994b). Defeasible logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pp. 353–394 Oxford. Clarendon Press.

Nute, D. (1997). Defeasible prolog, chapter 11. In Convington, M. Nute, D., & Vellino, A. (Eds.), *Prolog Programming in Depth, Second edition*. Englewood Cliffs, NJ. Prentice-Hall.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

Impossible Worlds and Logical Omniscience: A Note on MacPherson's Logic of Belief

Ahti Pietarinen
ahtivp@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract MacPherson (MacPherson, 1993) argues that the impossible worlds semantics does not provide a plausible framework for dealing with the problem of logical omniscience for epistemic logic, since it amounts to equivocation of logical connectives. It is shown, however, that MacPherson's own logic of belief, put forward to avoid logical omniscience, can nevertheless be embedded and interpreted in the impossible worlds semantics.

1 Introduction

It is well known that the logics of knowledge and belief, *i.e.* epistemic logics[1], formalised as extensions of the normal model system **K** and interpreted within the framework of "possible worlds semantics", are afflicted by a problem of logical omniscience: agents possessing epistemic attitudes towards propositions are ideal in the sense that they end up knowing all the logical consequences of their epistemic statements. Such epistemic systems are simply inconsistent with human rationality: no strain of reasoning power can make it possible for a human to know all the consequences of what he or she knows.

Sources for this unbouded rationality are easily detected, and certainly did not go unnoticed by Hintikka — the father of the epistemic logic — who in his book (Hintikka, 1962) coined the term "logical omniscience". Since the traditional possible worlds semantics treats knowledge as necessity and since every normal modal system contains the axiom **K**

$$\begin{array}{ll} \text{if } \models \varphi \text{ then } \models B\varphi & \text{(belief as necessity)} \\ B(\varphi \Rightarrow \psi) \Rightarrow (B\varphi \Rightarrow B\psi) & \text{(normality axiom } \mathbf{K}\text{)} \end{array}$$

it immediately follows that every formula that is implied by a formula that is known by an agent is also known. Let us formulate, thus, the property "logical omniscience" as follows.

Definition 1.1 *Whenever an agent a knows the formula* φ *(*$B_a\varphi$*), and* φ *logically implies the formula* ψ *(*$\varphi \models \psi$*) with respect to the class* $\mathbf{Mod}(\mathcal{M})$ *of models, then the agent a knows* ψ *(*$B_a\psi$*).*

There have been a number of attempts to overcome this problem. One of the earliest proposals was based on impossible (non-normal, non-classical) worlds (Hintikka, 1975; Rantala, 1982a, 1982b). In this approach, possible worlds are adjoined with impossible worlds, where not all the classical laws of logic hold. When evaluating her knowledge, an agent can conceive of and take into account impossible as well

[1] We will use knowledge and belief interchangeably throughout the paper, and subsume doxastic logic (logic of belief) under epistemic logic. Therefore, the term "logical omniscience" is used instead of the term "logical omnidoxasticity".

as possible worlds, for they are the usual epistemic alternatives abreast of possible worlds. However, the truth conditions in impossible worlds are not recursively defined, and hence can be completely free.

The method of impossible worlds semantics works. It solves the problem of logical omniscience for good, but the price is high. One must give up many of the intuitions that underpin traditional logicality, and familiar properties usually assigned to logical systems. In particular, functional completeness fails: the interpretation of the logical constants is not fixed. This is one of the main reasons why many researchers have been left dissatisfied with impossible worlds semantics. Among others, MacPherson (MacPherson, 1993) has recently argued that according to the impossible worlds semantics, the failure of functional completeness of logical connectives means that connectives are subject to equivocation. The connectives receive different interpretations in possible worlds and impossible worlds. As MacPherson himself puts it, "a connective cannot receive different interpretation in impossible worlds and remain classical". MacPherson goes on to represent his own system of belief which does not, so the story goes, suffer form omniscience, nor are the connectives any longer equivocating in different worlds.

In order to see pros and cons of both approaches, we shall first consider impossible worlds semantics in more detail and show how logical omniscience is disposed of in this approach. After that we describe MacPherson's own solution to the logical omniscience problem. We also show how different forms of omniscience can be dealt with in his logic of belief. Finally, we demonstrate how MacPherson's semantics can be subsumed under the impossible worlds semantics.

2 Logical Omniscience and Impossible Worlds

We have given only one but the most important and general form of logical omniscience. Traditional possible worlds semantics is, however, committed to many other closure properties as well, depending on the axiomatisation being used. The most common ones are:

LO1:	If $B\varphi$ and $B(\varphi \Rightarrow \psi)$ then $B\psi$	(closure under material implication)
LO2:	If $\models \varphi$ then $\models B\varphi$	(closure under valid formulas)
LO3:	If $\models \varphi \Rightarrow \psi$ then $\models B\varphi \Rightarrow B\psi$	(closure under valid implication)
LO4:	If $\models \varphi \Leftrightarrow \psi$ then $\models B\varphi \Leftrightarrow B\psi$	(closure under logical equivalence)
LO5:	If $B\varphi$ and $B\psi$ then $B(\varphi \wedge \psi)$	(closure under conjunction)
LO6:	If $B\varphi$ and $B\neg\varphi$ then $B(\varphi \wedge \neg\varphi)$	(closure under inconsistent beliefs)

Closure **LO1** represents the normality axiom **K** and closure **LO2** the rule of necessitation. Whether it is admissible to regard **LO1** as really representing some problematic aspect of omniscience depends on the axiomatisation, since most of the nonomniscient logics of knowledge and belief assume some weak axiom system which includes closure **LO1**. This is done in order to guarantee soundness, hence it is reasonable to assume that a class of agents must use modus ponens reliably. Nevertheless, it is clear that the full logical omniscience as in the definition 1.1 is avoided. We can also see the problematic feature of the above closures from the computational perspective: certainly it is not the case that an artificial agent could compute all logical consequences of its beliefs, for the agents are resource-bounded and under strictures of computation time and memory space.

The above closure properties hold in every logic of belief where belief is taken as analogous to necessity. An obvious but naive attempt to circumvent the problem of logical omniscience therefore, would be to relax the assumption and consider belief as a possibility instead of a necessity. This approach, however, is not satisfactory. Of the above properties, **LO2**, **LO3** and **LO4** still hold in the normal logic of belief where the possibility operator is construed as belief. Hence the omniscience definition 1.1 also holds, even though some of the weaker forms of omniscience may not.

Logical omniscience is certainly a property that has its roots deeper in the very way that knowledge

is defined in the possible worlds semantics, namely, as truth in all possible worlds. For this reason, much stronger variants of the basic semantic framework have been proposed, such as Hintikka-Rantala's impossible worlds semantics (Hintikka, 1975; Rantala, 1982a, 1982b).

An impossible worlds model is a 4-tuple $\mathcal{M} = \langle \mathcal{W}, \mathcal{W}^*, \rho, V \rangle$, where $\mathcal{W}$ is a non-empty set of possible worlds, $\mathcal{W}^*$ is a set of impossible worlds, ρ is an accessibility relation defined on $(\mathcal{W} \cup \mathcal{W}^*) \times (\mathcal{W} \cup \mathcal{W}^*)$, and $V : (\Gamma \times (\mathcal{W} \cup \mathcal{W}^*)) \to \{0,1\}$ is a valuation function which assigns truth values to all formulas Γ in all possible and impossible worlds. For all $w \in \mathcal{W}$ we have that

$$\begin{array}{lll} V(\neg\varphi, w) = 1 & \text{iff} & V(\varphi, w) = 0 \\ V(\varphi \wedge \psi, w) = 1 & \text{iff} & V(\varphi, w) = V(\psi, w) = 1 \\ V(B\varphi, w) = 1 & \text{iff} & V(\varphi, w') = 1 \text{ for all } w' \in \mathcal{W} \cup \mathcal{W}^* \text{ such that } w\rho w'. \end{array}$$

For all $w^* \in \mathcal{W}^*$ a detachment condition that validates the axiom **K** is imposed:

If $V(\varphi, w^*) = V(\varphi \Rightarrow \psi, w^*) = 1$ then $V(\psi, w^*) = 1$.

The key difference between the standard possible worlds semantics and the impossible worlds semantics is that the valuations in the impossible worlds are not recursively defined. A valuation V can behave in an arbitrary way in the impossible worlds. However, an agent must still consider impossible worlds when evaluating epistemic formulas and hence, depending on the choice of the impossible worlds and the valuations on these, may fail to believe the logical consequences of some of its beliefs. For example, closure under valid implication is not valid in the impossible worlds semantics, since although we have a valid formula $(\varphi \wedge \psi) \Rightarrow \psi$, the formula $B((\varphi \wedge \psi) \Rightarrow \psi)$ becomes invalidated simply by choosing an impossible world w^* such that $w\rho w^*$ and $V((\varphi \wedge \psi) \Rightarrow \psi, w^*) = 0$. Closure under conjunction is not valid either: consider $V(\varphi, w^*) = V(\psi, w^*) = 1$, and $V(\varphi \wedge \psi, w^*) = 0$. Continuing this way, the following proposition is easily verified.[2]

Proposition 2.1 *None of the previously given examples of logical omniscience holds in impossible worlds semantics.* □

Impossible worlds semantics thus solves all the problems considering logical omniscience. The solution is, however, rather trivial: no matter what closure properties are put forward, the truth conditions for the logical connectives in the impossible worlds can be chosen such that none of the closures can be generated. This is a very general method, and Wansing (Wansing, 1990) has in fact shown that many approaches suggested to mitigate or overcome logical omniscience can be subsumed under the impossible worlds framework. Moreover, Thjisse (Thjisse, 1992) has generalised the logic of general awareness of Fagin and Halpern (Fagin & Halpern, 1988) and proved that this generalisation which he has termed as *sieve semantics* is equivalent to the Hintikka-Rantala impossible worlds semantics.

The generality of the solution is not without a cost. In particular, functional completeness fails in the impossible worlds model. Suppose one is to add an extra definition of conjunction precisely as before:

$V(\varphi \,\&\, \psi) = 1$ iff $V(\varphi) = 1$ and $V(\psi) = 1$.

A strange consequence is now that even though definition of this "conjunction &" is completely analoguous to the previous definition, $B(\varphi \wedge \psi)$ is not equivalent to $B(\varphi \,\&\, \psi)$. They may have different truth values in some possible worlds, for the models are now different.

MacPherson refers to the strange behaviour as "equivocation of logical connectives", even though he is at pains to explain this obvious nonlogical feature of impossible worlds. Similar fears on the

[2] In the original semantics of Rantala, the detachment condition is imposed to guarantee soundness of the semantics; **LO1** cannot thus be rendered invalid. In some treatments of impossible worlds semantics this aspect is, for some reason, ignored (*e.g.* in (Hoek & Meyer, 1995, pp. 87–89,273)).

inadequacy of the original impossible worlds semantics have been put forward by Muskens (Muskens, 1992), who proposes type-theoretic solutions to them.

3 MacPherson's Logic of Belief

MacPherson's own suggestion to overcome logical omniscience is based on the notion of "partial descriptive alternatives". These are stipulated sets of propositions that can act as the agents' alternatives, thus retaining intuition of belief as a relation between an agent and alternative "scenarios". However, the notion of alternatives is adjusted such that they do not have to be complete structures like in the traditional possible world counterparts. Consequently, partial alternatives do not give rise to logical omniscience in the sense of definition 1.1. A formula ψ implied logically by a formula φ that is believed is not believed if, due an incomplete set of alternatives, it fails to be among those alternatives an agent conceives of as possible in the classical possible worlds model. The model of belief as a partial descriptions has its origins in Kripke's proposal (Kripke, 1980).

The language of MacPherson's logic of belief (BEL) consists of connectives $\neg$ and $\wedge$, a set of agents $a_i, i > 0$ and a belief operator B_{a_i}.[3] Moreover, BEL is modelled as a 3-tuple $\mathcal{M} = \langle S, g, V_{BEL} \rangle$, where S is a set of partial descriptions $\varsigma_i, i > 0$ (sets of propositions), g is a function that assigns to each a_i exactly one partial description $(g(a_i) \in \varsigma_i)$, and V_{BEL} is a valuation function as follows:

$$\begin{array}{lll} V_{BEL}(\neg\varphi) = 1 & \text{iff} & V_{BEL}(\varphi) = 0 \\ V_{BEL}(\varphi \wedge \psi) = 1 & \text{iff} & V_{BEL}(\varphi) = V_{BEL}(\psi) = 1 \\ V_{BEL}(B_{a_i}\varphi) = 1 & \text{iff} & \varphi \in g(a_i). \end{array}$$

In order to guarantee soundness, MacPherson imposes the following conditions on all partial descriptions:

(1) if $\varphi \in \varsigma_i$ and $\varphi \Rightarrow \psi \in \varsigma_i$ then $\psi \in \varsigma_i$;
(2) if $\varphi \wedge \psi \in \varsigma_i$ then $\varphi \in \varsigma_i$ and $\psi \in \varsigma_i$;
(3) for all $i > 0, (\varphi \wedge \neg\varphi) \notin \varsigma_i$.

The major point in the above definitions is the valuation rule for belief formulas. At partial descriptions, truth values are not given to the formulas. They merely are or are not the members of a given partial description. This point is cruicial, for the lack of the truth values of the formulas in ς_i now helps in preserving functional completeness: since logical connectives are interpreted extensionally in valuations, they do not acquire new interpretations in partial descriptions. Therefore, connectives are not equivocating, since there exists only one, that is, classical, interpretation for them.

MacPherson's BEL seems to do well with the logical omniscience problem. MacPherson does not, however, consider how well BEL fares with *all* the possible forms of the logical omniscience. It can, nevertheless, be seen that BEL dispenses with the closure properties:[4]

Proposition 3.1 *None of the previously given examples of logical omniscience holds in BEL.*

PROOF. Let us consider each example in turn.
LO2: Rule of necessitation does not exist at all.
LO3: Suppose $\varphi \Rightarrow \psi$ is valid. Choose $\varsigma_1 = \{\varphi\}, S = \varsigma_1$ and $g(a_1) = \varsigma_1$. Since $\varphi \in g(a_1), V(B_{a_1}\varphi) = 1$. But since $\psi \notin g(a_1), V(B_{a_1}\psi) = 0$ and therefore clearly $V(B_{a_1}\varphi \Rightarrow B_{a_1}\psi) = 0$.
LO4: Equivalent formulas do not have to be members of the same partial description.

[3] Originally, MacPherson allows his language to contain also constants and predicate variables. Without the loss of generality, we restrict our treatment to the propositional fragment of the original language.

[4] Here again, closure **LO1** is valid because of the given detachment condition.

LO5: Choose $\varsigma_1 = \{\varphi, \psi\}$ and $S = \varsigma_1$. Now $\varphi \in \varsigma_1$ and hence $V(B_{a_1}\varphi) = 1$. Similarly, $\psi \in \varsigma_1$ and hence $V(B_{a_1}\psi) = 1$. But $\varphi \wedge \psi \notin \varsigma_1$ and hence $V(B_{a_1}(\varphi \wedge \psi)) = 0$.
LO6: $B_{a_1}\varphi$ can be satisfied and $B_{a_1}\neg\varphi$ can be satisfied without satisfying $B_{a_1}(\varphi \wedge \neg\varphi)$, by choosing $\varsigma_1 = \{\varphi\}, \varsigma_2 = \{\neg\varphi\}, S = \{\varsigma_1, \varsigma_2\}$. Now, clearly $V(B_{a_1}\varphi) = 1$, since $\varphi \in \varsigma_1$ and $V(B_{a_1}\neg\varphi) = 1$, since $\neg\varphi \in \varsigma_2$. But since $(\varphi \wedge \neg\varphi) \notin \varsigma_i, i = 1, 2$ (it is also excluded by the condition (3)), $V(B_{a_1}(\varphi \wedge \neg\varphi)) = 0$. □

Both MacPherson's BEL and Hintikka-Rantala's impossible worlds semantics provide very flexible methods to dispose of logical omniscience. From the previous proof it can be seen that the main feature of how BEL copes with logical omniscience is to allow such partial descriptions which fail to include certain propositions as members. In distinction to the impossible worlds approach, the "accessible alternatives" are not given truth conditions and cannot thus equivocate. There is a sense, however, in which BEL can be understood in the impossible worlds framework.

Theorem 3.1 *Let $\mathcal{M}_{BEL} = \langle S, g, V_{BEL}\rangle$ be a model for the semantics of BEL and let $\mathcal{M}_R = \langle \mathcal{W}, \mathcal{W}^*, \rho, V_R\rangle$ be a model for the impossible worlds semantics. Then for every $\mathcal{M}_{BEL}$ there exists a model $\mathcal{M}_R$ such that $\mathcal{M}_{BEL} \models \varphi$ iff $\mathcal{M}_R \models \varphi$.*

PROOF. Let $\mathcal{M}_{BEL}$ be a model for BEL. Define $\mathcal{M}_R$ such that

$\mathcal{W}$	$=_{def}$	S, where S is the set of partial descriptions;
$\mathcal{W}^*$	$=_{def}$	$\{\varsigma_i \mid g(a_i) = \varsigma_i$ for some $i > 0\}$;
ρ	$=_{def}$	$\{\langle \varsigma_i, w_j\rangle \mid g(a_i) = \varsigma_i$ for some $i > 0, w_j \in R \subseteq S$ for some $j > 0, i \neq j\}$, where R is a subset of those sets of partial descriptions that hold according to the beliefs of an agent a_j.

Define then the mapping V_R: $(\Gamma_R \times (\mathcal{W} \cup \mathcal{W}^*)) \rightarrow \{0, 1\}$ such that

- for all $w \in \mathcal{W} \cup \mathcal{W}^*$: $V_R(p, w) = 1$ iff $p \in w$ (p is an atomic formula);
- for all $w \in \mathcal{W}$: $V_R(\neg\varphi, w) = 1$ iff $V_{BEL}(\varphi, w) = 0$; $V_R(\varphi \wedge \psi, w) = 1$ iff $V_{BEL}(\varphi, w) = V_{BEL}(\psi, w) = 1$; $V_R(B\varphi, w) = 1$ iff for all $w' \in \mathcal{W} \cup \mathcal{W}^*$, if $w\rho w'$ then $V_{BEL}(\varphi, w) = 1$;
- for all $w^* \in \mathcal{W}^*$: $V_R(B\varphi, w^*) = 1$ iff $\varphi \in w^*$.

Let us then prove by induction on the structure of φ:

$V_{BEL}(p, w) = 1$ iff $p \in w \overset{(def.)}{\text{iff}} V_R(p, w) = 1$;
$V_{BEL}(\neg\varphi, w) = 1$ iff $V_{BEL}(\varphi, w) = 0 \overset{(ind.ass.)}{\text{iff}} V_R(\neg\varphi, w) = 1$;
$V_{BEL}(\varphi \wedge \psi, w) = 1$ iff $V_{BEL}(\varphi, w) = 1$ and $V_{BEL}(\psi, w) = 1 \overset{(ind.ass.)}{\text{iff}} V_R(\varphi, w) = 1$ and $V_R(\psi, w) = 1$;
$V_{BEL}(B\varphi, w^*) = 1$ iff $\varphi \in w^* \overset{(def.)}{\text{iff}}$ for all $w^{*'} \in \mathcal{W}^*, V_R(B\varphi, w^{*'}) = 1$. □

It is to be seen whether the converse holds. It can be conjectured, however, that since in previous formulations of BEL and Hintikka-Rantala models all and the same instances of logical omniscience were invalidated, the two semantics are, in a sense, equivalent.

4 Conclusion

It can be argued that MacPherson's approach, in spite of its effectiveness, lacks the intuitive semantical appeal of the traditional possible worlds semantics. It seems to be syntax in semantic disguise. The very

basic intuition of knowledge or belief as a truth in all possible worlds, in all epistemic alternatives an agent can conceive, is not well retained. Belief is modelled with the alternative scenarios, but truth is no longer an applicable concept in alternatives. Moreover, beliefs are not even really modelled, but merely represented. It remains to be seen what interesting properties such beliefs have. Furthermore, convincing answers should be tried to find to the questions of whether the two semantics are equivalent and also, where the stipulated descriptive alternatives that represent agents' beliefs come from in the first place.

References

Fagin, R., & Halpern, J. (1988). Belief, awareness and limited reasoning. *Artificial Intelligence*, *34*, 39–76.

Hintikka, J. (1962). *Knowledge and Belief.* Cornell University Press, Ithaca, NY.

Hintikka, J. (1975). Impossible possible worlds vindicated. *Journal of Philosophical Logic*, *4*, 367–379.

Hoek, W. v. d., & Meyer, J. J. C. (1995). *Epistemic Logic for AI and Computer Science*. Cambridge University Press, Cambridge.

Kripke, S. (1980). *Naming and Necessity.* Harward University Press, Cambridge.

MacPherson, B. (1993). Is it possible that belief isn't necessary?. *Notre Dame Journal of Formal Logic*, *34*, 12–28.

Muskens, R. (1992). Logical omniscience and classical logic. *Lecture Notes in Computer Science*, *633*, 52–64.

Rantala, V. (1982a). Impossible worlds and logical omniscience. *Acta Philosophica Fennica*, *35*, 106–115.

Rantala, V. (1982b). Quantified modal logic: Non-normal worlds and propositional attitudes. *Studia Logica*, *41*, 41–65.

Thjisse, E. (1992). *Partial Logic and Knowledge Representation, Ph.D. Dissertation*. University of Tilburg.

Wansing, H. (1990). A general possible worlds framework for reasoning about knowledge and belief. *Studia Logica*, *49*, 523–539.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

A Co-Evolutionary Approach to the Call Admission Problem in Telecommunications

Anil Seth
anils@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract An important and difficult problem in the management of telecommunications networks is how good policies can be developed for admitting or blocking customer call requests for a network of limited server capacity. This paper describes a new co-evolutionary approach to this problem, which describes customers and servers as individuals within a 'technological ecology', and models their interaction using a modified version of the Iterated Prisoner's Dilemma (IPD). The model is also deployed to tackle the more complicated 2-class problem, in which there are two customer populations, each with different priorities and service requirements.

For both classes of problem, good results are achieved in terms of low call blocking rates, even in the face of system noise and server failure.

Although the present research is exploratory in character, it is proposed that this co-evolutionary approach could eventually be deployed in real-time in a telecommunications context, allowing a network to adapt automatically to changes and problems as they arise. More immediately, it is suggested that this kind of system could be used in simulation off-line, and combined with some degree of executive control in a hybrid system. Such a technology would be within reach of current engineering viability.

1 Introduction

1.1 Network Design, Network Management

In telecommunications, problems associated with network design and management are of central importance. The design problem concerns the development of an effective network, comprising of links with given bandwidths, so that it is capable of simultaneously routing traffic demand (see (Magnanti & Wong, 1984) for a comprehensive review). The management problem, in contrast, concerns controlling such networks once they have already been built, and the call admission problem is a standard problem in this latter area.

The basic call admission problem (see e.g. (Rose & Yates, 1996)) concerns a set of customers who require access to a routing network. Access to the network is gained through servers, and the customers compete for these servers, with system performance evaluated through the average blocking probability of calls. This problem assumes that the network design problem has already been solved, and concentrates instead on the admission of traffic to such a network.

An extension to this basic scenario is the 2-class problem in which there are two customer populations, each with different priorities, demand characteristics, and service requirements. The system

must then develop effective policies to deal with the more complex situations that can arise in these environments.

1.2 GAs and the Call Admission Problem

Rose and Yates (Rose & Yates, 1996) have demonstrated the utility of genetic algorithms with regard to the call admission problem. They discuss methods for evolving 'policies' for the servers which dictate whether they admit or refuse call requests. The policies that they evolve are *centralised* in the sense that they treat the entire network as a single functional entity, rather than as an agglomeration of potentially autonomous units. They present successful candidates that display near optimal blocking probabilities, and they consider the relative merits of different types of genotype encoding schemes for their policies.

Two criticisms of the above approach can nevertheless be levied. Firstly, the way in which an optimal policy is evolved for implementation across all the servers presumes that these servers have, and will always have, up-to-date, complete, and accurate information about the global state of the system (including the states of all the other servers). This is not only implausible in large networks, but renders the method difficult to extend, as the overall population of servers may well increase (or decrease) over time. The related (and perhaps more significant) problem is that if the system environment is noisy, or if servers become dysfunctional, then in a policy determined in advance that is reliant on globally coherent behaviour, there is no guarantee that system behaviour will robustly adapt to changing circumstances and continue to deliver effective management.

In the present paper, a co-evolutionary approach is developed which describes a telecommunications network as an artifical ecology, thereby devolving control and management to the level of the individual servers. In this way, policies for call admission and blocking are implemented on a purely local basis, with a globally coherent network policy emerging out of the combination of many local interactions. A major advantage of this approach (and hypothesis of this paper) is that since there is no complex centralised policy, neither network extendability nor network robustness in the face of server failure should present serious problems.

The approach advocated here, as with many other co-evolutionary investigations, takes as a starting point the model for interacting agents provided by the Iterated Prisoner's Dilemma.

2 Co-Evolutionary Foundations

2.1 The Iterated Prisoner's Dilemma

The Prisoner's Dilemma has long been established as a tool of great value in co-evolutionary investigations, (Axelrod, 1984)(Langton, 1995)(May, 1973). Essentially, it provides a framework for modelling non-trivial interactions between agents, where the maximisation of individual short term gain minimises the collective welfare, as illustrated by the following anecdote:

Imagine that you and an alleged accomplice have both been arrested, accused of a heinous crime. You are held in separate cells, and upon interrogation you can either *cooperate* by denying all knowledge, or *defect* by implicating your accomplice. You have no idea what your accomplice will do, but if you both cooperate, you will both be released (the reward, **R**), and if you both defect, then both of you will be jailed (the punishment, **P**). However, if you defect and she cooperates, then you will recieve a payoff (the temptation, **T**) and she will go to jail for longer (the sucker, **S**). But if she defects and you cooperate, then you yourself are the sucker. The paradox is thus evident, - in a single meeting you will always do best to defect, in doing so either recieving the monetary payoff or avoiding being the sucker. But of

	player 2 cooperates	*player 2 defects*
player 1 cooperates	1:R=3 2:R=3	1:S=0 2:T=5
player 1 defects	1:T=5 2:S=0	1:P=1 2:P=1

Table 2: Prisoner's Dilemma Scoring Table

course the logic is the same for your alleged accomplice, and if you both defect then you will both do worse than if you had both cooperated (see table 1).

Cooperation is thus unlikely to arise in a one-shot Prisoners Dilemma, but if players can meet time and time again, and retain some memory of previous interactions, then cooperation on any given move does become a rational strategy. It is this Iterated Prisoner's Dilemma (IPD) that forms the core of the present study.

Many researchers have used genetic algorithms to evolve strategies to play the IPD (see e.g. (Axelrod, 1984),(Langton, 1995)). In these studies, as in the present model, the genotypes comprise of binary character strings representing policies for playing the IPD, with the length of the genotype determining the number of preceding moves (the game history) upon which each individual can base its strategy. It has been repeatedly demonstrated that cooperative strategies can and do arise and persist in artifical ecologies populated by these evolving strategies.

2.2 The Modified IPD

The present paper uses a modified version of the IPD in order to model the interactions between servers and customers in a telecommunications network. The way in which the IPD model reflects the functional constraints of the call admission problem is a major contribution of this paper, but is predicated upon an extension to the basic paradigm that is not unique to the present investigation, concerning *preferential partner selection.*

In the standard formulation of the IPD, interactions are arranged in a very orderly fashion, usually in a 'round-robin' tournament where every individual interacts once with every other on each iteration. The principle of preferential partner selection removes this constraint by allowing individuals to have some control over who they interact with, and this extension can lead to the emergence of interesting new dimensions of emergent behavioural structure.

Stanley et al. (Stanley, D., & Smucker, 1995),(Ashlock, Smucker, Stanley, & Tesfatsion, 1994) have published extensively on the formation of *social networks* in an IPD context where agents can choose who they would prefer to interact with, and refuse overtures from partners they consider unsuitable. Choice and refusal is accomplished with reference to continuously updated expected payoffs that each agent maintains for every other agent in the population (Stanley calls this an IPD/CR mechanism). They demonstrate that cooperation is evolved rapidly in such circumstances and they discuss the emergence of various kinds of metastable networks displaying distinct patterns of connectivity.

In the present investigation, a variant of the IPD/CR mechanism is implemented so that servers can decide which calls to admit and which to refuse, and also so that customers can have some control over which servers they attempt to access.

2.3 Modelling the Call Admission Problem

In the simple call admission problem, a set of customers making call requests has to be accommodated by a set of servers. Thus, instead of a single population, we now have a bipartite world (servers and cus-

tomers), with a member of the customer population *not* representing a person with a phone, but instead a distribution node for a set of calls that must obtain access to a server network. These distinct populations interact with each other according to a model derived from the IPD/CR, but evolve separately.

The customers make offers to the servers, based on (initially equal) expected payoffs. The servers, who likewise have (initially equal) expectations of the customers, evaluate the offers they have received and accept a quota of the most preferable, refusing the rest. These refusals constitute part of the blocked call measure. The accepted offers are then played out as an interaction modelled by a single IPD iteration, with a defection by a server constituting a further blocked call. Defections by customers are not generally allowed[1]. And so it goes on - the expectations of the servers held by the customers are updated (and vice-versa)[2], and new rounds of offers are made and played out.

After a certain number of rounds, breeding takes place in the distinct populations to form the next generation, with offspring deriving from the most successful individuals (in terms of IPD score) constituting the new populations[3].

The central aspect of the server is then a combination of the genotype specifying the call admission policy (described in more detail in the following section), and a set of expected payoff values for each of the customers, which together determine whether calls are blocked or not.

Servers are also characterised by their *capacity,* specifying how many calls they can accept each iteration, and *status,* reflecting the operational state of the server. A finite probability can be set for servers to fail during the course of a given iteration, and the *down-time* (the number of subsequent iterations for which the server remains dysfunctional) can also be modified.

The customers also maintain expectations of the servers, as well as genotypes for playing the IPD. Customers are further defined by arrival rates, which can vary from iteration to iteration according to a probability distribution, reflecting the fluctuating call request profiles encountered in real networks. The customers *have* to make the required number of offers to the servers (as specified by the arrival rate) - they are not allowed to remain inactive (as is the case in the standard IPD/CR). And customers also demand particular service rates, such that high service rates require more server capacity than low service rates.

For the more complicated 2-class problem, customer priority can be modelled by different customer types delivering different scores (and penalties) during each IPD interaction, (through the use of modified versions of the scoring table, see table 2). The two customer classes may also have different arrival rate and service requirements.

With the situation set out as above, co-evolution should then lead to the servers adopting effective policies to maximise their individual fitnesses, which should require cooperation (as in the IPD/CR) and hence a decent solution to the call admission problem. The effects of failure, system noise, and their consequences for behavioural stability can then be assessed in this context.

[1] Complete customer cooperation was enforced by ignoring the customer genotype when it specified defection. However, when defection *was* allowed (by not ignoring the genotype), there was no appreciable difference in model behaviour (customers still cooperated most of the time anyway). Customer defections may have an interpretation in the model through representing the use of servers without subsequent payment.

[2] The following equation is used to update expecations:
$exp[i+1] = \gamma.exp[i] + (1-\gamma).payoff$
where 'γ' is a memory weight, and 'payoff' represents whatever IPD score is awarded.

[3] If customer defections are not allowed, then the customer strategies do not really evolve at all; they will always cooperate regardless of the constitution of their genotypes.

	server cooperates	*server defects*
type 1 customer coop.	c:R=3 s:R=3	c:S=0 s:T=5
(type 1 customer defects)	c:T=5 s:S=0	c:P=1 s:P=1
type 2 customer coop.	c:R=5 s:R=5	c:S=-2 s:T=8
(type 2 customer defects)	c:T=8 s:S=-2	c:P=-1 s:P=-1

Table 3: Call Admission Model Scoring Table

2.4 Genotype Encoding Scheme

The encoding scheme employed in the present model is an extension of a system employed by Lindgren[4] (Lindgren, 1991).

At the heart of each individual is a genotype, consisting of a string of `c`'s and `d`'s, which determine a strategy for playing the IPD. The longer the genotype, the more it can be influenced by the past history of interactions with other individuals - and the servers maintain separate game histories for each customer that they interact with (and likewise for the customers).

When the game is being played, each time a previous move is considered, half of the genotype is (temporarily) thrown away; one half if the move had been cooperative, and the other if it had been a defection. In this way, a genotype of length 16 can encode a strategy with a memory of 4 prior interactions, (after cutting a string of 16 characters in half 4 times, you are left with just a single character).

However, the genotype must be lengthened in order to specify the initial moves up until the memory limit is reached. A memory 4 strategy would require an extra 9 alleles to code for the initial 3 moves before the final 16 alleles can be used (thereafter, any number of moves can be made with the final 16 alleles determining the strategy). In the present model, a genotype length of 127 characters is used, providing a memory of up to 6 prior interactions.

Each time an iteration of the game is played, the moves made are stored in a large history array so that they can be accessed the next time the two players meet. In the present model this array is centralised, but in principle each server could maintain its own unique smaller array, as the servers only need to access those parts of game history in which they themselves took part.

The way in which any given server/customer interaction proceeds is therefore determined by a combination of the genotypes of the server and customer (which specify strategies for any possible game history of up to 6 prior interactions), and the actual history of interactions (if any) between the particular server/customer pairing that is being considered.

3 Implementation

The implementation of the model here bears certain similarities to the IPD/CR model of Stanley (Stanley et al., 1995), in that customers choose servers on the basis of the expected payoff values that they maintain[5]. But whereas in the standard implementation of IPD/CR the agents have to play as many rounds of IPD as they have tolerable offers pending, in the call admission model the total number of rounds playable is restricted by the server's capacity and the service requirements of the customer call requests. Thus the servers must rank the received offers and refuse (or block) the surplus.

Breeding is implemented with simple generational tournament genetic algorithms, but it is ensured

[4] Lindgren considered infinitely iterated IPD games with regard to the dynamics of species complexity, and used mathematical analysis to describe the behaviour of the ecology. The present model not only has completely different objectives, but also actually instantiates the 'ecology', therefore necessitating considerable alterations to Lindgren's scheme.

[5] Stanley, however, considers a single population rather than a bipartite populations of servers and customers

that the customer and server populations do not interbreed. Fitness is determined by the total score on the modified IPD, and it is the genotype coding for the IPD strategy (i.e. the call admission policy) that is modified in the breeding process. Population statistics are also calculated separately for each distinct population.

The main procedure of the model is implemented as follows:

```
randomly initialise bipartite populations
FOR EACH generation
    FOR EACH iteration
        servers fail according to the failure rate
        customers choose most preferable servers and
            make offers (according to variable
            arrival rates)
        servers rate offers and refuse least
            preferable ones that exceed capacity
            (taking account of different service
            requirements in the 2-class problem)
        FOR EACH tolerable offer for each server
            one round of IPD is played (with or
                without noise), and expectations
                are updated
        ENDFOREACH
        game history array and various scores are
            updated, blocking probabilities are
            calculated
        each server status evaluated and brought back
        on line if sufficient down time has elapsed
    ENDFOREACH
    new generations are created through bipartite
        breeding and tournament GAs
    every so often population statistics are
        calculated, and presented with graphics
ENDFOREACH
```

The model was coded in ANSI C and executed on a Sun Sparc workstation.

4 Results

4.1 Overall System Performance

The system typically evolves to very stable situations very quickly, with both customer and server cooperation very near to maximum all the time. This is a promising first observation as it suggests that even when control is completely localised and devolved, and even when short-term benefits can be gained through 'antisocial' behaviour, the system as a whole rapidly reaches an equilibrium beneficial to both customers and servers.

The patterns of connectivity at the beginning of each generation are unpredictable due to the initial expectations all being equal, providing no basis for partner choice. But as each generation wears on, stable patterns of interaction develop, - the same customers being admitted by the same servers, with occasional alterations. This stability persists over many generations.

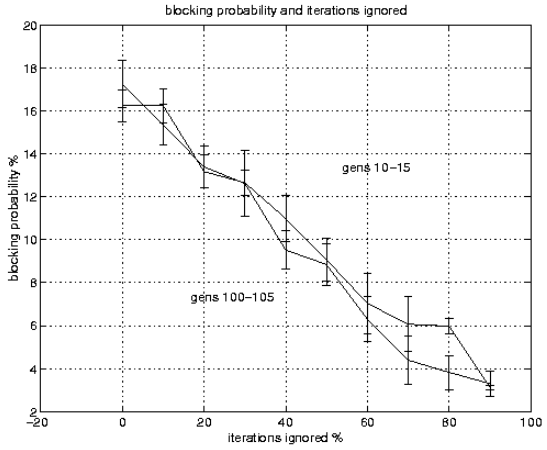

Figure 1: *the blocking rate falls off as each generation progresses (the horizontal scale determines the percentage of the earlier part of the generation that is igonred when calculating the statistic).*

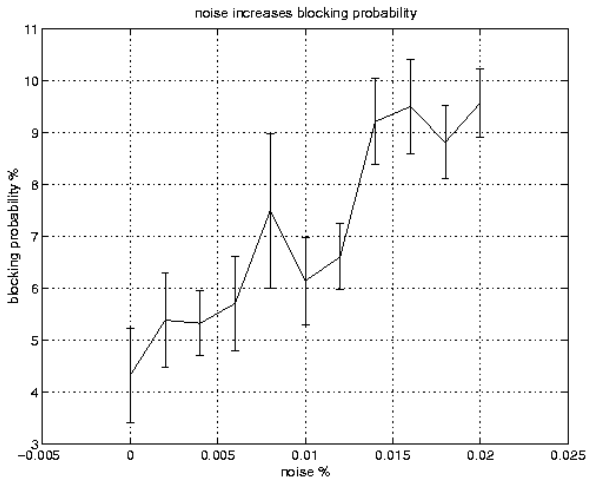

Figure 2: *noise degrades performance on the call admission problem, but in a non-catastrophic fashion.*

4.2 An Easy Problem

The model was initially explored with a very simple problem - with 20 customers (of a single type) and 10 servers, allowing each server to deal with 2 customers at each time step. With neither noise nor server failure, the blocking probability was typically very stable and very low. However, since the customer/server expectations are reset at the beginning of each generation, the blocking probability was always highest at these times, and would then fall off as the generation progressed. Figure 1 illustrates how the blocking probability is lower if earlier iterations are ignored (in a given generation), and suggests that in a steady-state system, without generational upheavals, the actual blocking probability would be very low indeed. Figure 1 also illustrates that the situation after 100 generations is very similar to that after just 10, implying that evolution acts very quickly to deliver a stable situation.

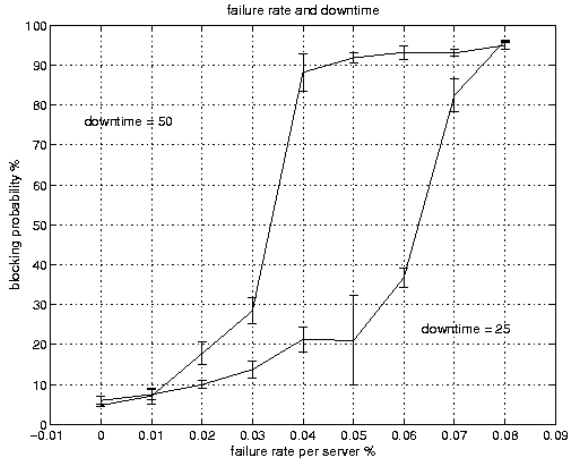

Figure 3: *low levels of server failure are successfully accommodated by the model, but high levels of server failure lead to system collapse - the critical point is dependent upon server down-time.*

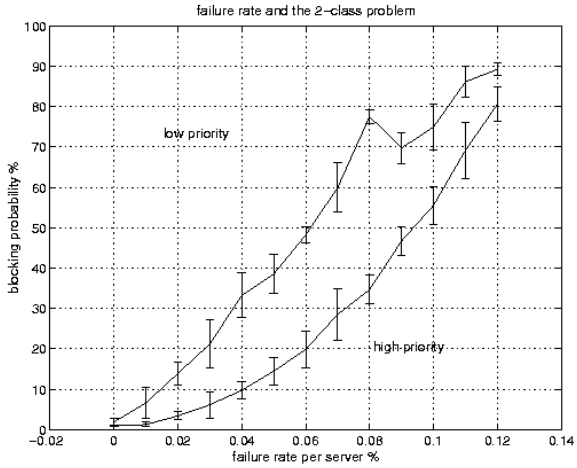

Figure 4: *low priority customers suffer more than high priority customers when servers fail in the 2-class problem.*

4.3 Noise and Failure

The effects of noise and server failure were also explored in the context of the simple problem outlined above, with noise referring to the probability with which the opposite action to that specified by the genotype of the server or customer is performed. In most cases this simply means the probability with which calls are accidentally blocked (although it is also the probability with which blocked calls are accidentally processed). Figure 2 shows that whilst system performance is degraded with increasing noise levels, this degradation is not catastrophic.

Figure 3 examines server failure, demonstrating that with low failure rates the system dynamically reconfigures the flow of call admissions in order to cope - but after a critical stage is reached this process breaks down and almost all calls are blocked. This critical point is dependent upon the server downtime; the longer each server is down for, the earlier the system performance collapses.

4.4 The 2-class Problem

Many runs were executed with various different parameters set for the relative numbers of high and low priority customers, their respective call arrival rates, service rates, and server capacities. In all cases, the general policy emergent in the system was to process all high priority calls, and then as many low priority calls as possible. If calls had to be blocked, the low priority customers suffered (even when high priority calls required more server capacity).

The preference for high priority customers is undeniably a rational policy, and is also evident in how the system reacts to server failure. Figure 4 illustrates that as server failure becomes more predominant, the low priority calls suffer more than those of high priority.

4.5 Extendability

The ability of the system to adapt well to extension was assessed through the introduction of additional servers. Ten extra servers were introduced after half the total number of generations had elapsed. The new servers were genotypically initialised randomly, just like the original population - and all had equal expected payoff values for all the pre-exisiting customers (as the customers themselves did for all the new servers). Nevertheless, the system proved able to dynamically reconfigure itself in order to take advantage of the extra capacity afforded by the new servers.

Figure 5 illustrates a single customer class call admission problem, in which the initial capacity is insufficient to admit all the call requests. After 50 generations, 10 new servers are introduced and the call blocking rate immediately shrinks. There is also a notable spike in the diversity of server genotypes as the initially random genotypes of the new servers come into play.

In the 2-class problem, a lack of sufficient server capacity is initially manifest in a high blocking rate for low priority calls (high priority calls are still getting through). With the addition of new servers, the difficult situation for the low priority calls is considerably ameliorated, as illustrated in figure 6.

5 Conclusions

The model described in this paper is possibly one of the first pure Alife models that has been blooded for a telecommunications application. The success enjoyed suggests that co-evolutionary/game-theoretic approaches to technological management, - either through the instantiation of completely autonomous interacting agents or in simulation as part of a hybrid system - herald a bright future.

In the context of the call admission problem, the application of the modified IPD/CR delivers good results in terms of call blocking probabilities and, in all cases, both servers and customers evolve highly cooperative strategies. Environmental noise proved detrimental to system performance, but in a non-catastrophic way. The system was also able to maintain stability in the face of server failure - although after a critical point blocking rates soared to nearly 100 percent (dependent on parameters such as down-time). Similarly, dynamic reconfiguration enabled the system to take efficient advantage of additional servers if they were introduced at later stages. No executive interference was necessary for this process to take place. This effectively demonstrates a major advantage of a continually evolving distributed system over a preprogrammed centralised system, in which central control programs would normally have to be updated carfeully in order to maintain effective management over any new situations.

With two customer classes, a simple policy of processing all high priority calls and as many low priority calls as possible was adopted, across a wide range of parameters. This is certainly a rational, and perhaps the *most* rational policy to adopt in these kinds of circumstances. And with server failure, the low priority calls bore the brunt of the extra call-blocking much harder than the high priority calls.

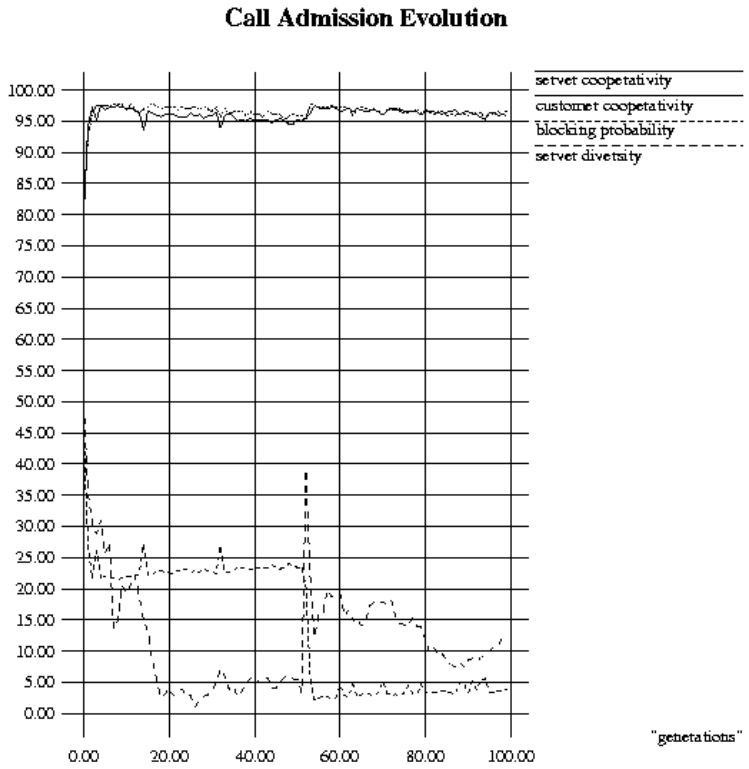

Figure 5: *introduction of extra servers leads to dynamic reconfiguration of the system and a concomitant reduction in the blocking rate.*

However, the introduction of additional servers, in all cases, ameliorated the blocking of these low priority calls.

In sum, the model presented here delivers a new, robust, and effective principle for developing and deploying call admission strategies. The main question which then arises concerns the relationship that this model should have with existing and near-future telecommunications technology. There are two primary alternatives. In principle, the system could be located on-line, with call admission strategies evolving in real-time in a real network. This would be a very strong interpretation of the work presented in this paper. Alternatively, the system could be used in simulation off line, and combined with a certain minimal degree of central control in a hybrid system.

There are two further possibilities here. The most simple and weakest interpretation would be to use simulations solely to guide designers of network technology in their quest to understand how networks might behave in all kinds of circumstances, much as car designers can now use virtual wind tunnels. A more interesting possibility would be for the policies evolved off-line to be periodically downloaded into the real-world system, and for real-world changes to be uploaded into the simulation. In this way, executive control can still be maintained over the global behaviour of the system, whilst allowing most of the advantages of the co-evolutionary system to manifest themselves, albeit with some small time lag.

One point must be raised in order to qualify the conclusions laid out above. Whilst it is true that the co-evolutionary approach does eliminate the need for executive control over the policies implemented by the system, it is *not* true to say that executive interference is *completely* abolished by this method.

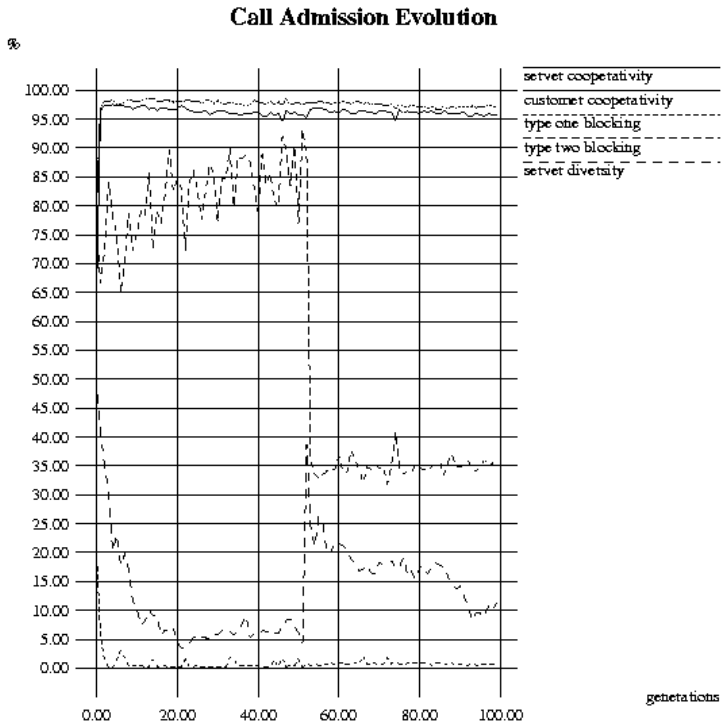

Figure 6: *introducing extra servers reduces the high blocking rate experienced by low priority customers in the 2-class problem.*

Both the system of offers and refusals, and the implementation of the genetic algorithm itself, require the synchronised and centrally co-ordinated interaction of individuals within the system. Practically, this is unlikely to present a problem, and indeed may even be of benefit in that it provides another way in which overall central control can be maintained without diminishing the benefits that accrue from this kind of system.

6 Future Work

6.1 Integration with Executive Control

Integration with a central controller is an obvious avenue for development, and would help bring ecological technology (if of a diluted variety) within the domain of engineering viability. Hybrid systems of the type mentioned in the previous section would provide a suitable direction for implementation.

6.2 Reinforcement Learning

Considerable interest has recently arisen in the possibilities afforded by the implementation of reinforcement learning techniques in telecommunications (see e.g. (Littman & Boyan, 1993), perhaps since this provides another way of incorporating individual-level adaptive techniques without the abdication of all central control. The ecological call admission model itself makes use of reinforcement learning, in the way in which the servers and customers update their expected payoff values over time. Thus, learning theory could well provide a fruitful territory for bridging the chasm between centralised and ecological control methodologies that would reward further exploration.

6.3 Lamarckian Inheritance

Lamarckian inheritance could be introduced into the system, so that expected payoffs are not periodically reset but are passed down from parents to children. This would suggest the use of a continuous GA instead of a generational GA, and would clearly alleviate the problem of the surge in blocking rates at the start of each new generation.

Acknowledgements

This work owes a great deal to my supervisors Phil Husbands and Inman Harvey, and also to Tony White of Nortel Ltd. This work was funded in part by the ESPRC and in part by Nortel Ltd.

References

Ashlock, D., Smucker, M., Stanley, E., & Tesfatsion, L. (1994). Preferential partner selection in an evolutionary study of prisoner's dilemma. Economics report 35, Iowa State University.

Axelrod, R. (1984). *The Evolution of Cooperation*. New York : Basic Books.

Langton, C. (1995). *Artificial Life; an Overview*. MIT : Bradford Books.

Lindgren, K. (1991). Evolutionary phenomena in simple dynamics. In Langton, C., Farmer, J., Rasmussen, S., & Taylor, C. (Eds.), *Artificial Life II*. Addison-Wesley.

Littman, M., & Boyan, J. (1993). A distributed reinforcement learning scheme for network routing. In *Proceedings of the First International Workshop on Applications of Neural Networks to Telecommunications*, pp. 45–51.

Magnanti, T., & Wong, R. (1984). Network design and transportation planning: Models and algorithms. *Transportation Science*, *18*, 1–55.

May, R. (1973). *Stability and Complexity in model ecosystems*. Princeton University Press, Princeton, NJ.

Rose, C., & Yates, R. (1996). Genetic algorithms and call admission to telecommunications networks. *Computers and Operations Research*, *23*(5), 485–499.

Stanley, E., D., D. A., & Smucker, M. (1995). Iterated prisoner's dilemma with choice and refusal of partners: Evolutionary results.. In Moran, F., Moreno, A., Merelo, J., & Chacon, P. (Eds.), *Advances in Artificial Life : Lecture Notes in Artificial Intelligence*. Springer-Verlag.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

Estimation of a Probability Distribution over a Hierarchical Classification

Diana McCarthy
diana.mccarthy@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract Many natural language processing applications make use of hierarchical classifications whilst also having a statistical framework which requires an estimation of the probability distribution over the taxonomy. Data for estimating the probability distributions typically comes from corpora but estimation is complicated by the ambiguity of the data. One application involving such a task is the automatic acquisition of selectional preferences. The method of estimating these class probabilities is crucial to the success of the technique and this paper compares the previous schemes and concludes that correct modelling of class inclusion is essential. Modifications to previous approaches are described which help to curb the effect of ambiguity and avoid overly-general results.

1 Introduction

The automatic acquisition of lexical information is an active area for current NLP research. The selectional preferences that predicates have for their arguments are one such valuable piece of information. Such work usually relies on a hierarchical classification of word senses and this has typically involved the lexical thesaurus WordNet (Miller, Beckwith, Felbaum, Gross, & Miller, 1993b).

WordNet is a lexical resource for open class words of English which is organised by part of speech (verb, adjective, adverb and noun). Within these categories all entries are defined in terms of word senses rather than word form, i.e. it is a thesaurus rather than a dictionary. WordNet entries (classes or synsets) are sets of synonymous word senses which are related to other classes of the same POS category by relations such as hyponymy. Any noun may belong to more than one class and any class may have more than one member. To illustrate the noun *"chicken"* belongs to three classes. The first sense is that of **MEAT** the second sense **BIRD** and the third **WIMP** [1]. These are illustrated along with some synonyms and hypernyms in figure 1.

This paper concerns the issues raised when counting occurrences of nouns in data and attempting to estimate the frequency distribution over word senses of these nouns by designating the appropriate WordNet class or classes for each noun token. For example which classes should we increment when we see the word *"chicken"*. How should the frequency (and therefore probability) distribution be divided between the various classes with direct membership and those with indirect membership? A probability distribution should sum to 1 but how should the probabilities at hypernyms relate to the probabilities of their hyponyms? This paper describes how these questions are tackled in two schemes for estimating probability distributions over the WordNet noun hypernym hierarchy and describes two modifications to one of these schemes which reduce the effect of ambiguity and thereby reduce over-generalisation.

2 Previous Work

Previous work estimating the probability distribution over the WordNet has typically involved the noun hypernym hierarchy for the acquisition of selectional preferences. There are two main approaches used, that by

[1] In diagrams I represent WordNet classes by some of the words stored at the class in WordNet. Meanwhile in text I use one of these close synonyms (if available) or a salient word at a hypernym class

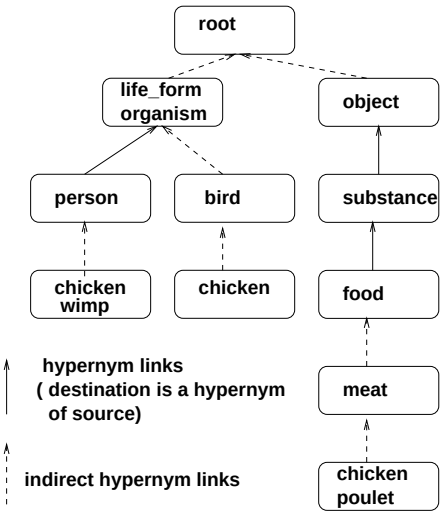

Figure 1: Hypernym hierarchy showing the senses of "chicken".

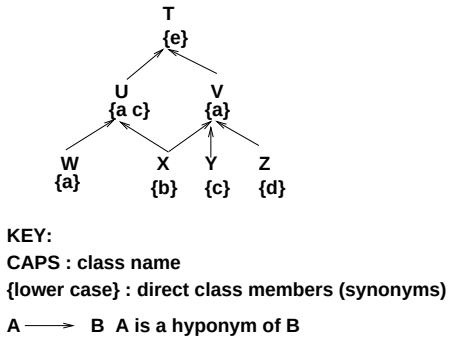

Figure 2: Taxonomy A.

Resnik (Resnik, 1993) and that by Ribas (Ribas, 1995b, 1995a), and Li and Abe (Li & Abe, 1995; Abe & Li, 1996). The description of how the class probabilities are calculated differs between Li and Abe and Ribas but the methods are essentially the same.

To illustrate the difference in the approaches we will consider taxonomy A in figure 2. The lower case letters represent direct membership of the classes (in UPPER CASE) under which they appear. The arrows indicate the hypernymy relationship and any lower case member of a hyponym is an implicit indirect member of the hypernym classes. In this way the class **T** has *"e"* as a direct member and *"a b c d"* as indirect members. A class may have more than one member e.g. **U**, and an item may belong to more than one class e.g. *"a"*.

Resnik's approach doesn't distinguish between hypernymy and polysemy when estimating the frequency distribution. The frequency of a noun, in a given sample, contributes to all classes the noun belongs to, regardless of direct or indirect membership. Furthermore each frequency count is divided by the number of these classes to ensure that the sum of probabilities over the entire hierarchy equals one, equation 1 describes his estimation of the frequency of a class (c).

$$freq(c) = \sum_{n \in nouns\_at\_or\_under(c)} \frac{1}{|classes(n)|} \times freq(n) \quad (1)$$

Table 4: Resnik's Frequency and Probability Distributions.

CLASS	FREQ	PROB = $\frac{Freq}{5}$
T	$\frac{2}{4}+\frac{1}{4}+0+\frac{1}{3}+\frac{1}{1}=2.0833$	$0.41\dot{6}$
U	$\frac{2}{4}+\frac{1}{4}+0=0.75$	0.15
V	$\frac{2}{4}+\frac{1}{4}+0+\frac{1}{3}=1.08\dot{3}$	$0.21\dot{6}$
W	$\frac{2}{4}=0.5$	0.1
X	$\frac{1}{4}=0.25$	0.05
Y	0	0
Z	$\frac{1}{3}=0.\dot{3}$	$0.0\dot{6}$

In the work of Resnik and all other works described in this paper the estimation of class probabilities from the class frequencies is the straightforward maximum likelihood estimate:

$$\hat{p}(c)=\frac{freq(c)}{N}, where\ N=\sum_{c'\varepsilon all\_classes} freq(c') \tag{2}$$

The crucial flaw in Resnik's scheme arises from the lack of distinction between direct and indirect membership. This means that the probabilities of hyponym classes are not subsumed by their hypernyms. This also gives rise to a number of anomalies because the contribution of a noun depends on the depth of the classes (Ribas, 1995a).

For example, given the taxonomy A in figure 2, if the string *"a b d a e"* is observed the frequency and probability distributions would be as shown in table 4. The frequency contribution from each "word" (lower case letter) is shown separately at each of the classes to which it belongs, directly or indirectly. The class probabilities in Resnik's scheme sum to 1 but there are anomalies that Resnik himself acknowledges. For example **X** and **Z** both have members (*"b"* and *"d"* respectively) which occur the same number of times (once) but the classes end up having different frequency counts because of the difference in the number of hypernyms above them. Resnik suggested in his thesis that the assignment of class probabilities warranted further attention.

Ribas, in contrast, wants to adhere to the maxim that the probabilities of all the possible senses of a word should sum to one rather than the probabilities of all senses and hypernyms. In this case the frequency of a class must be divided by the number of classes in which the word has direct membership. Additionally the probability of a hypernym should be the sum of the probabilities of all hyponyms plus any probability it has by virtue of direct membership. In this way the probability at the root of the hierarchy should equal one. He formalises this as a weight that is applied to the frequency count for each noun. The weight is specific to the noun (n) and class (c) and is the ratio between the number of classes directly containing n beneath and including c and the total number of classes containing n, this is shown in equation 3 and is equivalent to Ribas' estimation of the conditional probability of a class given the noun.

$$weight(n,c)=\frac{|direct\_classes_n\_at\_or\_under\_c|}{|direct\_classes_n|} \tag{3}$$

The unweighted frequency count at any class from any noun belonging at or beneath the class is simply the number of occurrences of that noun in the corpus. This is then multiplied by the weight as in equation 3. The frequency of a class is then the sum of all the weighted frequencies of nouns in the corpus belonging by direct or indirect membership:

$$freq(c)=\sum_{n\varepsilon nouns\_at\_or\_under(c)} freq(n)\times weight(n,c) \tag{4}$$

The frequency and probability counts for the example string *"a b d a e"* are shown in table 5. The frequency estimate from each letter is again shown separately. The contribution of *"a"* is shown as the first component of the addition for the frequency estimation of each class it belongs to. It belongs to classes **W**,**U**,**V** and **T**. For each of

Table 5: Ribas' Frequency and Probability Distributions.

CLASS	FREQ	PROB = $\frac{Freq}{5}$
T	$(2\times\frac{3}{3})+\frac{1}{1}+0+\frac{1}{1}+\frac{1}{1}=5$	1
U	$(2\times\frac{2}{3})+\frac{1}{1}+0=2.\dot{3}$	$0.4\dot{6}$
V	$(2\times\frac{1}{3})+\frac{1}{1}+0+\frac{1}{1}=2.\dot{6}$	$0.5\dot{3}$
W	$2\times\frac{1}{3}=0.\dot{6}$	$0.1\dot{3}$
X	$\frac{1}{1}=1$	0.2
Y	0	0
Z	$\frac{1}{1}=1$	0.2

these classes the unweighted frequency 2 is multiplied by the appropriate weight. The denominator of the weight is 3 in all these cases, the number or polysemes of *"a"*. The numerator however is 1 for **V** and **T** since only one direct sense falls under these classes. Meanwhile for **U** and **T** the numerators are 2 and 3 respectively reflecting the membership at and under these classes.

The basic scheme for frequency assignment used by Li and Abe is effectively the same as that of Ribas but just expressed a little differently. The frequency estimate for a class is initially calculated only with direct membership in mind using the sum of the frequencies of each noun belonging at that class divided by the total number of classes this noun directly belongs to. These frequencies are then cumulated up the hierarchy ensuring that a hyponym only contributes its frequency once to each hypernym (this being necessary because WordNet is a DAG with multiple inheritance). The frequency estimation, and therefore probability estimation, is in effect the same as that of Ribas but the contributions of direct members are calculated by a separate process than those of indirect members, this is shown by equation 5.

$$freq(c) = \sum_{n \in nouns\_at\_class(c)} \frac{freq(n)}{|direct\_classes_n|} + \sum_{n \in nouns\_under\_class(c)} \frac{freq(n)}{|direct\_classes_n|} \quad (5)$$

With either formulation the results are the same. As can be seen in table 5 the probability of the root T equals 1. Also, this method ensures that for any noun the sum of probabilities $p(c|n)$ at the direct classes equals one whereas in Resnik's scheme it is the sum of probabilities from all classes for a noun (direct or hypernyms) that equals one.

3 Modifications to Improve Estimation.

The methods for estimating the class probabilities proposed by Ribas and Li and Abe make sense because they take the hypernym relation into account in a different way to the handling of polysemy. Hyponym probabilities are correctly assumed to be subsumed by hypernyms which accords with the class inclusion assumptions associated with a hypernym or "is a" relationship. Furthermore the frequency distribution given a particular noun is sensibly divided between the possible senses of that noun, and not the potentially large number of hypernyms of the noun. Li and Abe's approach for calculating the probability distribution and acquiring selectional preferences is taken as a starting point. Two modifications are made to the basic approach.

In this work selectional preferences are obtained largely as in Li and Abe's work (Li & Abe, 1995; Abe & Li, 1996). The preferences are represented by a cut across the WordNet noun hypernym hierarchy where the class members of the cut are not ancestors of one another and each is assigned a score indicating the strength of the preference. These models are known as association tree cut models (ATCMs) as the scores indicating preference are given by the association score in equation 6, the log of which gives the measure known as mutual information. This measures the "relatedness" between two words, or in this class-based work on selectional preferences between

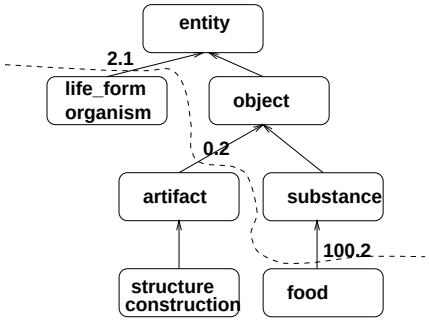

Figure 3: ATCM for 'eat' object slot.

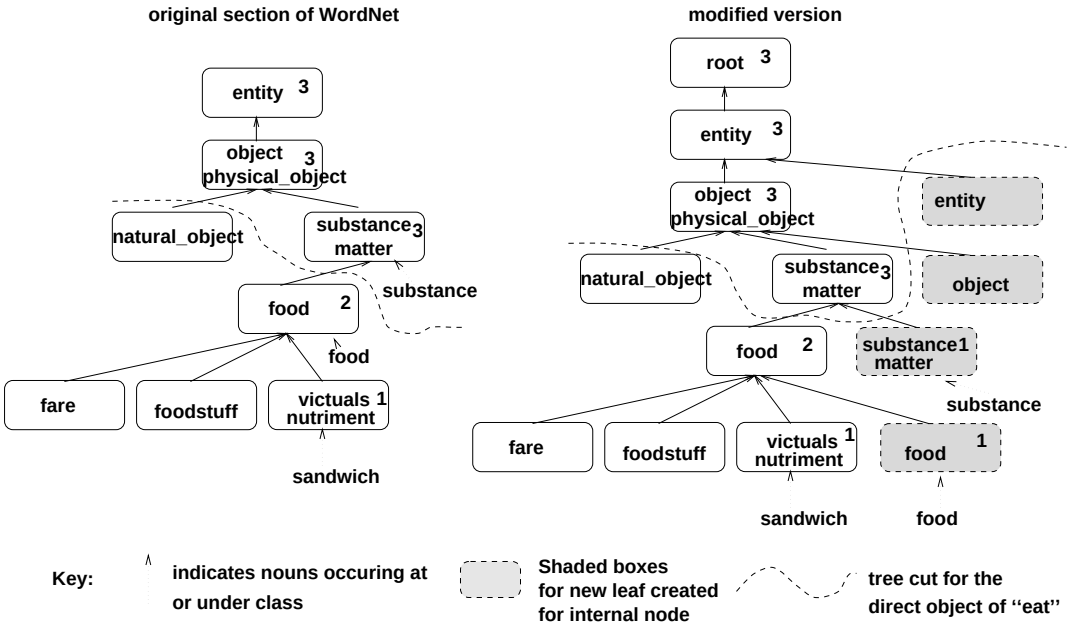

Figure 4: Creating New Leaves for Internal Nodes.

a class (c) and the verbal predicate (v).

$$A(c,v) = \frac{p(c|v)}{p(c)} \tag{6}$$

For example an ATCM for the direct-object slot of *"eat"* might look like that in figure 3.

The method of calculating class probabilities follows that of Li and Abe except for two important departures which are described in the two following sub-sections.

3.1 Modifications to WordNet

Li and Abe's method of obtaining selectional preferences requires a hierarchy where all words (terminals) appear on leaf classes. The method relies on finding cuts across the tree where no members of the cut are related by hypernymy and where they partition the set of words exhaustively and disjointly, i.e. all words appear at or under these classes on the cut. The sum of class probabilities on any such tree cut will equal 1. All the classes in

Table 6: Frequency by depth of classes with multiple-inheritance

Depth	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Freq	0	0	1	3	22	111	161	147	68	21	12	11	0	1

WordNet's hierarchy contain noun word forms as members so in order to meet this requirement Li and Abe prune the tree at all classes where they have observed an occurrence in the data. This means that a far shallower version of WordNet is used. This however can give rise to overly-general preferences when the data contains an argument which occurs higher in the hierarchy than the prototypical arguments. For example, if a direct object such as *"entity"* in the predicate argument pair "build:entity" is observed then pruning the hierarchy at this level would result in loosing detail on the types of entities that are built (e.g. **OBJECT** vs **LIFE_FORM**).

In contrast, here the tree is not pruned but new leaves are created beneath each internal node. The frequency count attributed to nouns listed at the internal node is instead placed at the new leaf and then cumulated up the hierarchy in the usual way.

To illustrate look at the transformation of a small portion of the hierarchy in figure 4. The class frequency distribution for three heads, *"substance"*, *"sandwich"* and *"food"*, from the direct object slot of *"eat"* is shown by the numbers in the right-hand corner of the class boxes [2]. All classes without numbers have 0 frequency with respect to this small sample. In the unmodified version of WordNet it is quite possible to have a cut which does not cover all the noun senses and so the probability distribution along it would not sum to one. This is because terminal nouns such as *"substance"* occur at internal classes which can occur above a candidate cut. In our scheme the frequency, and therefore probability, contributions from such terminals are moved down to newly created leaves and so the probability axioms are maintained. In contrast the strategy adopted by Li and Abe would be to prune the tree at the **SUBSTANCE** class which would result in a shallow tree with no possibility of distinguishing between a preference for **FOOD** or **DRINK** at the direct object slot of *"eat"*.

One outstanding problem with the structure of WordNet arises because WordNet is actually a DAG rather than a tree. This is acknowledged by Li and Abe but nothing is done to circumvent this problem. This is also true of our implementation. We have not as yet attempted to compensate for this fact because of the small extent of the problem. On examination of the cases of multiple inheritance we find that less than 1% of the classes in the noun hypernym network have more than one parent class. Moreover, the majority of such classes are deep down in the network, as can be seen from table 6, and many concern compound nouns e.g. "school boy" "head nurse" which currently are not handled by our system. In cases of multiple-inheritance the probability distribution is still valid in that any hypernyms above more than one parent of a class will have the frequency count incremented only once for the multiply inherited class. Thus, in the single case at depth 3 pictured in figure 5 a frequency count of 1 detected at the **PERSON** node will only be incremented once at the superordinate **ENTITY** class. A cut at this class will therefore meet the probability axioms. A cut at the layer of **CAUSAL_AGENT** and **LIFE_FORM** would however give rise to the sum of the classes on a cut including these to be greater than 1. This however is understandable when we consider that multiple-inheritance gives rise to overlap between the two parent classes both in terms of the subsumed nouns senses and their probability. The result is a coherent probability distribution across the cut. [3]

3.2 Word Sense Disambiguation

The second modification concerns the pervasive problem of word sense ambiguity. All the researchers whose work is discussed here acknowledge the problem and hope that the effect will be reduced with enough data. However Li and Abe and Ribas all report interference from erroneous senses. Li and Abe try and get rid of some of this interference by placing a threshold on class probabilities before including the classes in the cut representing selectional preferences. Even though a certain amount of noise can be removed we expect that resolution of the

[2] For the purposes of this example assume that these words are monosemous for the sake of simplicity

[3] Although the probability distribution is valid even in cases of multiple-inheritance there are unresolved problems arising because the method used to obtain the most appropriate cut across WordNet relies on a tree structure and not a DAG. These issues have been left for further research

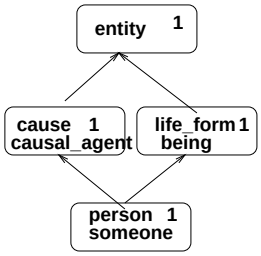

Figure 5: An example of multiple-inheritance.

ambiguity would improve the results.

A method was sought for sense tagging the argument heads that did not require a large amount of training time (unsupervised approaches) nor manual effort to create large amounts of training data (supervised approaches). The approach selected was that where the most frequent sense is chosen regardless of context (Wilks & Stevenson, 1996). Criteria on when to apply the most frequent sense is specified and if these were not met then the ambiguity was left unresolved as in the work of Li and Abe and Ribas. The criteria included two thresholds, one on the frequency of the predominant sense and the other on the ratio between the first sense and the second. The details are described in (McCarthy, 1997). Although supervised training material is required for estimation of the sense frequencies this is smaller than would be the case with context dependent methods since fewer parameters are involved. There is already a portion of the Brown corpus that has been manually tagged with WordNet senses for the SEMCOR project (Miller, Leacock, Tengi, & Bunker, 1993a) and this was used for the sense frequency data.

To indicate the differences in estimation of class probabilities suppose we have an occurrence of the verb direct-object pair "eat:chicken" in our corpus, what would the class probability distribution look like? There are three senses of the noun *"chicken"*. These are indicated in figure 1. In Resnik's scheme a frequency count of 1/20 would be added to all direct classes containing *"chicken"* as well as all hypernyms. This is because there is a total of 20 such classes. In contrast Li and Abe and Ribas would place 1/3 at each direct sense and then these would be incremented up the hierarchy. Thus the probability distribution for the noun *"chicken"* over its direct senses would sum to one. In the scheme we are advocating, the predominant sense of *"chicken"* meets our criteria in the SEMCOR data (its first sense occurs 16 times whilst the second sense occurs 7 times). Thus the frequency 1 would all be placed at a new leaf (since this is an internal node) under the **MEAT** sense of *"chicken"*.

This is helpful because in the case of *eat:chicken* the predominant sense is the correct one. There are however occasions when the disambiguation might work against us for example given an occurrence of the verb direct-object pair *eat:fish* the predominant sense of fish is the **ANIMAL** sense rather than the **MEAT** sense. The rationale for using this rather rudimentary form of sense tagging is that it will help more than it hinders and so the overall results will be improved.

3.3 Experimental Results

ATCMs were obtained using the subcategorization acquisition system described by Briscoe and Carroll (Briscoe & Carroll, 1997). This system was used to construct lexicons which list argument heads at the appropriate slots. Two sets of data were used both from shallow parses of the British National Corpus. The first was from 1.8 million words of parsed text and which formed a subset of the 10.8 million words used to construct the second lexicon. The point of trying two sample sizes was to see if the benefits of the crude disambiguation diminished with the larger sample. Evaluation was performed both by informal observation and quantitative evaluation comparing the acquired preferences to data from a "gold standard".

The ATCMs were obtained for a test set of 30 verbs:
add, agree, allow, ask, begin, believe, bring, build, call, cause, change, charge, choose, consider, cut, decide, end, establish, expect, feel, find, fix, give, help, like, move, produce, provide, seem, swing.

These verbs were selected because of the range of complement patterns they exhibit and were not picked with respect to their selection properties.

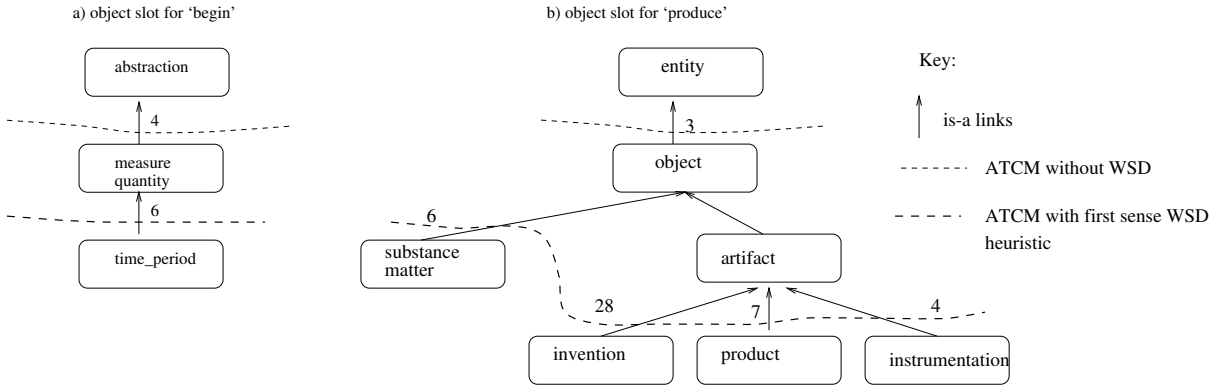

Figure 6: ATCM for a) 'begin' object slot and b) 'produce' object slot.

3.4 Informal Evaluation

From informal observation it could be seen that calculation of class probabilities using the crude disambiguation resulted in more appropriate (and usually less general) cuts than those where no disambiguation was used. Over-generalisation is cited as a result of noise from erroneous senses (Ribas, 1995a).

This can be seen in figure 6 which shows a small part of two ATCMs for the direct object slots of *"begin"* and *"produce"* respectively. These were produced from the larger lexicon built using the 10.8 million word sample. For clarity only a few classes on the cuts are pictured, these are classes involving strong preferences. The ATCMs obtained using the first sense heuristic are indicated with a bolder line and fall below the ATCMs produced without this heuristic. This specificity often coincides with intuition for example the direct object slot of *"begin"* features a strong preference for the class **TIME_PERIOD** when the first sense heuristic is used whilst without this heuristic a weaker preference is indicated for the hypernym class **MEASURE**. In the case of *"produce"* a preference is shown for the class **OBJECT**, without any disambiguation of the input data, whilst with the first sense heuristic the preference is more distinctive indicating preferences for *"inventions"* whilst not for nouns such as *"bridge"* belonging to the **STRUCTURE** class. In addition to the specificity the first sense heuristic tends to produce fewer but stronger preferences than without WSD, though this needs to be quantified.

An even more striking difference was shown with the smaller lexicon from 1.8 million words. With the smaller sample of argument heads the cut for some verbs was at the root indicating that no preferences could be observed for these verbs. This occurred more frequently without word sense disambiguation affecting 12 of the target verbs as opposed to 8 when the first sense heuristic was used. [4]. All 30 verbs had ATCMs below the root node with the larger data set.

The improvement observed informally is supported by some preliminary evaluation.

3.5 Gold Standard Evaluation

The "gold standard" used here is the Cambridge International Dictionary of English (CIDE) (Procter, 1995). The dictionary, as with most others, provides alongside each entry a list of example uses. From these the head nouns at direct object slot position have been obtained manually and each assigned a WordNet class that best represents the sense of the noun. As much as possible these senses have contained the noun in their set of synonyms but in cases where an appropriate class cannot be found as close a sense as possible to the meaning of the head noun is chosen.

The preferences for the target verb have been evaluated against this set of test direct object senses. Each test item is scored correctly if it falls at or under one of the classes on the cut with an association score greater than 1. A baseline has been formulated that attempts to test the chance that a given sense will fall at or under a preference (class on the ATCM with association score above 1). This baseline takes an average of the ratios between the

[4]Preference acquisition was performed on only 28 verbs from the smaller sample as 2 had frequencies below a threshold set at 20.

Table 7: CIDE Evaluation

Data (Million)	1st S	% correct	BL	diff
1.8	No	53	37	16
1.8	Yes	70	46	24
10.8	No	92	58	34
10.8	Yes	89	48	41

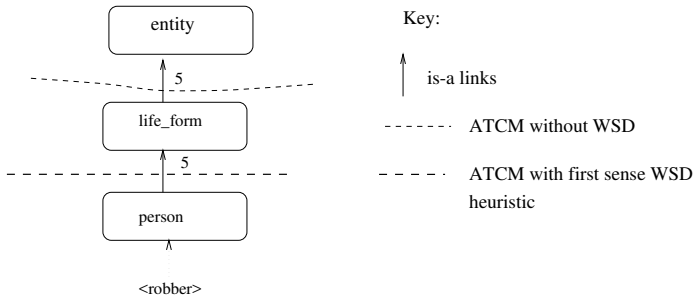

Figure 7: 'robber' under ATCMs for 'believe' object slot.

number of classes with a score above 1 and the number of classes on the cut for all instances in the test set. This is expressed in equation 7 where i is an instance from the test sample S, and v is the verb specified in the instance i.

$$\sum_{i \epsilon S} \frac{|prefs_v|}{|classes\_in\_ATCM_v|} \tag{7}$$

This baseline does not account for the specificity of the cut explicitly but a more specific cut will typically have more classes and less preferences (but stronger ones). It simply assumes that all target senses will fall under the cut and measures the chance that an item will fall under one of the classes on the cut that expresses a preference.

For example, the verb : direct object pair *"believe:robber"* was observed in the dictionary. The sense assigned for *"robber"* was that under **PERSON**. This falls under both ATCMs with and without the first sense heuristic as can be seen in figure 7. Without disambiguation the baseline ratio for this instance is $\frac{12}{25}$ whereas with the first sense heuristic the ratio is $\frac{9}{26}$.

Results are shown in table 7. The last column indicates the difference between the % correct and the baseline (BL). For both sample sizes the difference is increased by the first sense heuristic (indicated by 1st S).

From both informal inspection and preliminary evaluation these modifications do appear to improve the selectional preferences obtained.

4 Summary

All the methods for acquiring selectional preferences from hierarchies such as WordNet are dependent on a good estimation of class probabilities. Methods should respect the class inclusion relationship of hypernyms for their hyponyms. Additionally care needs to be taken when distributing the frequency of polysemous nouns. The frequency distribution for any particular noun should be spread over its direct classes in the absence of any means of disambiguation. The problem of word sense ambiguity is so extensive that although with large samples some noise can be removed there is evidence that even crude sense disambiguation helps matters. It is predicted that a more detailed disambiguation, taking context into account, would improve accuracy still further but the pay-off would have to be worth the additional costs.

References

Abe, N., & Li, H. (1996). Learning word association norms using tree cut pair models.. In *Proceedings of the 13th International Conference on Machine Learning ICML*. Unpublished. cmp-lg/9605029.

Briscoe, T., & Carroll, J. (1997). Automatic extraction of subcategorization from corpora.. In *Fifth Applied Natural Language Processing Conference.*, pp. 356–363.

Li, H., & Abe, N. (1995). Generalizing case frames using a thesaurus and the MDL principle. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing*, pp. 239–248 Bulgaria.

McCarthy, D. (1997). Word sense disambiguation for acquisition of selectional preferences. In *Proceedings of the ACL/EACL 97 Workshop Automatic Information Extraction and Building of Lexical Semantic Resources for NLP Applications*, pp. 52–61.

Miller, George, A., Leacock, C., Tengi, R., & Bunker, R. T. (1993a). A semantic concordance.. In *Proceedings of the ARPA Workshop on Human Language Technology.*, pp. 303–308. Morgan Kaufman.

Miller, G., Beckwith, R., Felbaum, C., Gross, D., & Miller, K. (1993b). *Introduction to WordNet: An On-Line Lexical Database*. ftp//clarity.princeton.edu/pub/WordNet/5papers.ps.

Resnik, P. (1993). *Selection and Information: A Class-Based Approach to Lexical Relationships.* Ph.D. thesis, University of Pennsylvania.

Ribas, F. (1995a). *On Acquiring Appropriate Selectional Restrictions from Corpora Using a Semantic Taxonomy.* Ph.D. thesis, University of Catalonia.

Ribas, F. (1995b). On learning more appropriate selectional restrictions.. In *Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics.*, pp. 112–118.

Wilks, Y., & Stevenson, M. (1996). The grammar of sense - is word-sense tagging much more than part-of-speech tagging?. cmp-lg/9607028. Submitted to Natural Language Engineering.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

Software reuse from an external memory: The cognitive issues of support tools

Fabrice Retkowsky
fabricer@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract While early reuse techniques were based largely on the programmer's memory, more recent techniques give the programmer access to a library of existing programs or models. The problem arises of how to use these 'external memories': that is, how to structure the memories and their access methods effectively.

To study this problem from a cognitive point of view, we have to know how the programmer deals with programs, and more precisely, what knowledge a programmer has of a program when he reuses it. There are different models of those internal representations of programs. This paper looks at how programs are internalized (i.e. how this knowledge is build), and how programmers externalize their representations of programs.

The problem is made more difficult by the fact that software reuse is made of four steps (finding, understanding, specializing and integrating a component), which differ slightly from one reuse technique to another, and which involve different cognitive processes. Each step can be supported by a specific tool, and knowing which cognitive processes are involved should help us in optimizing those tools.

We suggest that a modular reuse system should be developed. The aim of this system would be to make comparisons between different theories on some important aspects of software reuse.

1 Introduction

Software reuse is an activity which includes many different reuse techniques, each one having its own specific properties, though, we can isolate a trend in the evolution of these techniques.

Early reuse was based on the programmer's memory: the basic principle was to recall a program written in the past, and then use it as a first approximation of the solution of the new problem. The major limitation of this kind of reuse is the programmer's memory. By giving access to a database of existing programs or models (such as a source code component library, a design component library, or a book of classical programs), this limitation can be removed: a programmer can then look at a large number of programs, and choose which one may be useful.

Yet the use of an external memory raises some cognitive issues. For example, how easy it is for the programmer to find a program that may be useful and then modify it for its new use.

In the first section of this paper we describe the most important reuse techniques, highlighting the distinction between internal memory based reuse and external memory based reuse. We then look at which important cognitive concepts are involved in the reuse activity. More precisely, we describe what is known of the internal and external representations of programs, as well as the internalization and externalization processes.

We see in the third section that reuse activity can be split into four different steps: finding, understanding, specializing and integrating a component. As such, we examine how each one of these steps, which can slightly vary from one reuse technique to another, relate to those four essential cognitive concepts. Finally, we study how far the designers of existing support tools have tried to tackle those cognitive issues and offer guidelines for the development of new tools.

2 From internal reuse to external reuse

Software reuse is a general term that includes many different techniques. For example, consider the case of a programmer who needs to program something that he (or somebody else) has already programmed, or alternatively who needs to program many instances of more or less the same program. He can either write a first version of the program, and copy/paste it where necessary, or scavenge some part of an existing program, or reuse a function which he found in a library, etc. Each one of these alternatives is a reuse technique of its own.

Here we will briefly describe the most common reuse techniques, by dividing them in two categories: internal memory reuse and external memory reuse.

2.1 Internal memory reuse techniques

2.1.1 Write/Copy/Paste

In this situation, the programmer has to write several instances of more or less the same program. He decides to write the first instance and then to copy/paste it for further instances, modifying it if necessary.

Détienne (Détienne,) describes this as 'new' reuse, as opposed to scavenging 'old' code. She suggests that the programmer first builds a high-level mental representation of the generic solution, and implements it in the first instance. Then, considering both the mental (high-level) representation and the first (low-level) instance, he implements a second instance, and so on.

The Write/Copy/Paste technique has not been studied a lot yet, possibly because of its small-scale aspect.

2.1.2 Code scavenging

The Code Scavenging technique is used when a programmer copies a part of the code of an existing program, modifying it if necessary. Hoadle (C. M. Hoadley & Clancy, 1995) describes Code Scavenging as 'code cloning', as opposed to 'code invocation' (using code components, see 2.2.1) or 'patterns' (design pattern components, see 2.2.3).

In fact, we can make a distinction between two kinds of code scavenging. On the one hand, the programmer may reuse a program that he has written himself in the past, based on his memory or the listing.

On the other hand, the programmer may be given from the beginning a program to reuse that he did not write himself. This situation is usually the one studied in experiments, but also occurs in real life, for example when a programmer has to completely rewrite an existing program. In this situation, the programmer does not know anything from memory of the program, but has one hopes some kind of documentation, as well as the code listing itself.

2.1.3 Design scavenging

Rather than reusing code, programmers sometimes reuse code abstractions, for example when they just remember the structure of a program they wrote in the past. They can then decide to reuse the same general methodology (the design) without looking at the code itself (the implementation). For instance, Krueger (Krueger, 1992) noticed that sometimes "a large block of code is copied, but many of the internal details are deleted while the global template of the design is retained". Then, "the developer can directly reuse these abstractions in the new design".

We can make a distinction between two situations. On the one hand, the programmer may simply remember how he solved a particular problem in the past, and decide to reuse the same method. This can be seen as a 'memory' situation. On the other hand, he may have a large piece of code to reuse, and not reuse it with all its precise details, but just reuse the design. This can be seen as a 'listing' situation.

2.2 External memory reuse techniques

As we said before, external memory reuse is characterized by the use of some organized stores of specially designed, reuse-oriented components. Such a store can for example be a database of reusable components, or simply a book of example programs. McIlroy's (McIlroy, 1968) was the first to propose an industry of 'off-the-shelf' source code components.

2.2.1 Source code component reuse

Source Code Component Reuse (SCCR) deals with the reuse of code component, which are stored in a library. Library, once again, has a broad meaning, including databases of pieces of code, books of example programs, etc.

But the basic principle is that some software engineers first have to write the code components whose basic aim is to be reused. For example, they can be designed as part of a current project. Instead of writing a piece of code specific to their own problem, software engineers may sometimes realize that they can make it reusable in other situations, by other programmers, and decide to write a reusable component instead. Alternatively, some organizations may setup a group of programmers whose task will be to look at the software produced by the whole organization, and try to turn some of it into reusable artifacts, by putting them into a library and documenting them.

2.2.2 Software architectures

The principle of software architectures is that it is possible to map high-level problem descriptions or designs to existing architectures or implementations. By looking at some existing systems, inside a strictly defined domain, it is possible to identify the most common architectures. Then, by describing one's problem, an automated system should be able to select the most suitable architecture, to adapt it to one's particular needs, and to compile it automatically into a complete implementation. However, this software architecture reuse technique, described by Neighbors (Neighbors, 1984) (Neighbors, 1989) and Krueger (Krueger, 1992), has not been adopted widely, perhaps because to design such a software architecture proved to be very difficult.

As a matter of fact, SAs now refer to the complete designs of some existing system, which can be adapted to a new problem, yet without the idea of an automated compilation process. Only the structure of the system is reused. As far as large systems are concerned, programmers consider that what is more important is the design of the system. From a well conceived design, the implementation is straightforward.

2.2.3 Design patterns

Design Patterns (DPs) are high-level representations of common aspects of designs. A pattern is a small structure of a few objects, with an accurate name and description.

Gamma, Helm, Johnson and Vlissides (E. Gamma & Vlissides, 1995) made a list of 23 basic patterns, where each pattern is defined by about ten attributes. These attributes describe the structure of the pattern, the issue addressed by this structure, and the consequences of integrating the pattern into a system. They underlined the fact that the aim of design patterns is to "program to an interface, not an implementation" and to "favor object composition to class inheritance".

Design Patterns may succeed were SCCR failed. The original idealistic view of software reuse was based on the concept of 'building blocks', but the code level proved to be too complicated to allow this. By contrast, Design Patterns are in theory very simple to use: it is basically a 'selecting then assembling' task. Therefore, it seems possible that DPs could allow reuse to be much closer to the original 'building block' ideal. That probably explains why DPs have been so successful amongst researchers, and have focused so much interest.

As we have seen, the restriction of the programmer's own memory has been removed by developing reuse techniques which are based on external memories. Now the problem arises of how to use those external memories. While early reuse naturally involved the programmer's thoughts and his internal memory, we are now in a situation where the programmer has to relate to an external device which extends or even replaces his internal memory. This may lead to some sort of disruption of the reuse activity, and we should study how this gap can be reduced.

3 Internal and external representations of programs

Considering that the issue of using an external memory involves such concepts as memory, perception and understanding, it seems logical to study this problem from a cognitive point of view. Four major points should be studied:

- *Internal representations of programs*

  We need to know what knowledge a programmer has of a program when he reuses it. This includes identifying the knowledge itself as well as understanding its structure and organization.

- *External representations of programs*

We also need to look at how programs can be described and documented, e.g. using the code listing, textual descriptions, or diagrams.

- *Internalization*

  Besides, we should look at how programs are internalized, that is how the internal representations are built. This may depend on the task the programmer is performing, and of course on the nature of the program, as well as its external presentation.

- *Externalization*

  In parallel, we should look at how programmers externalize their internal representations of programs, be it for programming or to describe a program. This may give us some information on the programming knowledge of programmers and on which external representations of programs they naturally use.

3.1 Internal representations

We can make the distinction between two types of programming knowledge: general programming knowledge, which is used to write, debug and understand programs, and knowledge of past programs, that is, internal representations of programs that are being (or have been) used.

3.1.1 Programming knowledge

An influential view, including the work led by Soloway (Soloway & Ehrlich, 1984), suggests that programming knowledge is made of two components:

- *Programming plans*

  These are small, language independent, units of code that perform a single basic task. For example, looking through a list of numbers is a programming plan.

- *Programming discourse rules*

  These rules guide the programmer on how to put different plans together. For example, by putting a 'compute the average' plan into a 'look through a list of numbers' plan, a programmer can compute the average of a list of numbers.

As a programmer grows in expertise, he learns more and more programming plans and discourse rules. However, Davies (Davies, 1995) showed that expertise does not only come from the amount of knowledge. Where intermediate programmers differ from novice programmers on the amount of plans and rules they know, experts programmers differ from intermediate programmers by the structure of their knowledge.

Indeed, experts can identify in a programming plan what is called a focal line. This focal line is thought to be the most important one, the line that sums up what the plan is doing. For example, in a 'compute the average' plan, which adds up different numbers and divide the total by the number of numbers, the focal line is the *average = total / number* line.

This theory of programming knowledge was based on Pascal-like languages. This supposedly includes most of the high-level languages (C, C++, Java, Ada, etc.). Some studies showed that the existence of plans for Basic or Fortran is not obvious.

3.1.2 Internal representations do exist

While the existence of general programming knowledge is a clear fact, the existence of internal representations of programs written by the programmer in the past has been a controversial issue. In this paper we make the hypothesis that programmers do have some internal representations of their programs. For example, it is clear that programmers are able to recall aspects of previous programs that they have written and perhaps the techniques they employed to derive these programs.

The work of Hoadley, Linn, Mann and Clancy (C. M. Hoadley & Clancy, 1995) tends to support this hypothesis. They conducted an experiment where subjects were asked to look at some functions, before solving a few problems where they could in fact reuse some of these functions. Reuse occurred more often when the subjects were previously asked to summarize the functions. Reuse was even more probable if they happened to summarize them in an abstract way. This suggested that programmers indeed build some internal knowledge of programs when they try to understand them, those representations being more or less abstract.

3.1.3 Models of internal representations

Before looking at some models of internal representations of programs, it is interesting to notice that memory can be characterized as episodic or semantic (Ormerod, (Ormerod, 1990)). While episodic memory stores information about particular events, semantic memory stores global, general, abstract information. For example, general programming knowledge, which is either learnt in an abstract form or abstracted from different situations, is commonly considered as being stored in semantic memory. Alternatively, we can wonder whether internal representations of past programs are semantic or episodic knowledge. It would appear natural to regard code-level representations as largely episodic in nature while design-level representations are in part semantic in nature.

Semantic networks Semantic networks are a representational system of long-term memory. In semantic networks, knowledge is symbolized by 'cells', which hold one precise topic (or word, meaning, object, etc.). These cells are linked together by many links which denote relations between topics, and which can be activated. Thus, starting from one topic (bird), we can go to other topics (animal, feathers, to fly), and again to other topics (plane), and so on.

Ormerod (Ormerod, 1990) describes how schema based knowledge (such as programming plans) might well be seen as being based on semantic networks. Since a plan is a piece of abstract, semantic knowledge, it may be associated to a particular node. The question is whether the same thing can be said of internal representations of programs, which as we have said before includes not only semantic but also (and mainly) episodic knowledge. This, as far as we know, remains to be studied.

The internal pseudo-language of experts One way of trying to understand a programmer's internal representations of programs is to look at the most instinctive, natural, elementary way in which he expresses them.

Petre (Petre, 1990) described how programmers "solve problems [...] in a private, pseudo-language that is a collage of convenient notations from various disciplines, both formal and informal", which "might be taken as the surface reflection of the expert's computational model". A program expressed in this language seems to be an "assemblage of overlapping and possibly incoherent fragments". This sounds quite similar to the idea of abstract, language-independent programming plans assembled by various programming rules.

As it is, this could suggest that programmers indeed represent programs internally using plans and rules. But the problem here is that these remarks are more about how programmers create programs (using plans and rules as we said in 3.1.1), rather than how their internal representations (such as their memories of past programs) are structured.

Program understanding von Mayrhauser and Vans developed, as we will see in 3.3, a complex theory of program understanding. According to this theory, programmers switch between three internal models (the top-down model, the program model and the situation model) when trying to understand a program. Therefore, we may think that these three internal models may be some kind of internal representations of programs.

But again, the problem is that this is basically a description of the knowledge that a programmer has of a program when he tries to understand it. But will he use the same internal representations when he remembers or recalls a program he has worked on in the past?

The fact is that, for now, we do not know a lot about internal representations of programs. There are many conjectures and hypotheses, but proven facts are rare. We would particularly need more information about the internal representations of old programs.

3.2 External representations

Here we present some external representations of programs, and how they can be accessed.

3.2.1 External representations

Code Code listing is the basic way of representing a program. It is the representation actually used to create the program, and therefore the best known by programmers. Code can be 'enhanced' by adding some more

information by putting spaces between different functions, indenting each instruction blocks to the left, putting keywords in a different colour, and so on.

Textual description Text can be added to the code listing, by way of embedded comments or on separate pages. This allows the programmer to really describe not only his program, but also the requirements, analysis and design.

Graphical representations Many graphical representations have been developed to support program description. They range from simple diagrams to complex data-flow or control-flow diagrams, as well as structure diagrams. Besides, many alternative sets of more or less meaningful and intuitive symbols exist. But the problem is that, for now, no study has proven that one of these graphical representations brings some clear benefit over code or text description. As Green (Green, 1990) said, "there are many more diagrammatic possibilities being touted by their supporters, than there are investigations".

3.2.2 Access techniques

Browsing Browsing was developed to navigate through large documentations, be they made of code, textual and/or graphical descriptions. The idea is that, by clicking on a word, the user can go to a part of the description which is related to this word. This system is usually called Hypertext, and it is widely used in the World Wide Web on the Internet. Furthermore, a search tool can be provided, so that the user can find out where one particular word is used in the documentation.

Intelligent agents von Mayrhauser and Vans (von Mayrhauser & Vans, 1995) highlighted the fact that, since program understanding seems to be based on hypothesis testing, a program description system should include the possibility for the user to ask questions, i.e. to test hypotheses. This can be considered as a plea for intelligent agents to be implemented in such systems. The agent would be able to help and guide the user through the documentation, and to assist him in understanding the program. Ideally, the user could feel like he is interacting with the original programmer himself.

3.2.3 What should be displayed?

A lot of different types of information can be provided using text. Brooks (Brooks, 1983) made a list of possible aspects to describe, not without noting that more description is not always better. Similarly, many different diagrammatic representations exist, such as control-flow, data-flow and structure diagrams. Their efficiency still remains to be proven. Though this issue will be studied more thoroughly in the following sections, we can recall two systems of representation which have been developed to describe programs.

The Description Level Green, Gilmore, Blumenthal, Davies and Winder (Green, 1989) developed the cognitive dimensions theory, which defines a collection of different cognitive aspects of programs, such as viscosity, premature commitment and perceptual cueing. One of their conclusions is that object-oriented programming systems, instead of relying so much on code and object relationships, should include a Description Level. This Level could include the Facets approach (Diaz and Freeman, (Prieto-Diaz & Freeman, 1987)), which gives a list of aspects upon which classes and functions can be described, and the Use of Multiple Views, which takes into account other relationships than simple code relations (e.g. class inheritance), such as chronological and design-based ones.

The integrated meta-model Alternatively, we can consider again the theory of understanding developed by von Mayrhauser and Vans (von Mayrhauser & Vans, 1995). As we said, it is based on three internal models of programs, and they argue that the information relevant to each one of these models should be provided, as a presentation system "should support the cognitive processes of understanding, not hinder it". Though, they don't explain how this knowledge should be externalized, that is which external representations should be used.

As we have seen, there are many different ways of presenting information. Though no study has yet proven that one given external representation has a definite overall advantage over the others, it seems logical to think that

there are cognitive differences between them.

In fact, what seems the most important at first is to find what precise knowledge should be displayed. Then, a precise study could be led to compare how various representations can display this type of information. The theories of Green et al. and von Mayrhauser and Vans could lead to such experiments.

3.3 Internalization

The internalization of programs is basically program comprehension, that is, the process of understanding a program using different sources, i.e. different external representations.

Soloway (Soloway & Ehrlich, 1984) argued that programming plans and discourse rules play a strong role in program comprehension. For example, he showed that expert programmers often make strong assumptions about programs, since they think that they all follow some common discourse rules.

By contrast, Pennington (Pennington, 1987) explains that programming plans just play a role in program comprehension and explanation, but that they are not the memory structure. The role of plans in internal representations of programs has not been, in her opinion, proven. As we see it, this confirms the distinction we made between general programming knowledge (which indeed seems to involve plans and rules) and internal representations of programs.

3.3.1 Models of program comprehension

Brooks Brooks (Brooks, 1983) described a model of program understanding based on hypothesis testing. To understand a program, the programmer has to make a global hypothesis on what the program does. Then he can split this hypothesis into sub-hypotheses, and so on, until the hypotheses attain a level of detail which make them comparable to the documentation available, e.g. code, texts, etc. A low-level hypothesis can thus be confirmed or disconfirmed by the documentation, in which case this hypothesis and its neighbours will have to be modified. This model is particularly focused on domain knowledge, or more precisely at how the program deals with the domain knowledge. Domain knowledge helps the reuser in formulating the hypotheses, and as a consequence, most hypotheses will be targeted at learning this domain knowledge.

Top-down model This model, described by Soloway, Anderson and Ehrlich, is quite similar to Brooks' one. The originality comes from the fact that they identify three types of programming plans: strategic plans (global methods used by the program to achieve a goal), tactical plans (local methods used to achieve a local goal) and implementation plans (language dependant implementations of a tactical plan). Here the focus is on goals and plans since the internal representation of the program is a decomposition of the overall goal in different plans, which are in fact local goals that are achieved by other sub-plans, and so on.

Bottom-up model The theory developed by Pennington is based on the building of two internal models of programs: the program model (which is a control-flow model abstracted from the code by spotting plans and independent chunks) and the situation model (which is a data-flow abstraction based on hypotheses which are checked on the program model). The reuser first begins to build the program model by analyzing the code. Then he can start building the situation model, which requires some domain knowledge. It is completed once the top-level hypothesis, that is the program goal, is reached.

Integrated meta-model von Mayrhauser and Vans (von Mayrhauser & Vans, 1995) built a theory which integrates the top-down model, the program model, and the situation model. Its fourth basic component is the knowledge base of the reuser. During the comprehension process, the reuser unconsciously switch from one model to another depending on what he has just found or understood.

We can see that these models are focused on code understanding, without looking at design understanding. Besides, if some provisions are made for the use of different types of documentation, they do not actually address this issue.

They confirm that program understanding depends on the subject's task (maintenance, debugging or reuse will not require the same level of understanding) and on his knowledge of the domain and of the program itself.

Finally we can notice that what results from the comprehension process is an initial internal model of the program, which is stored in short term memory. It may well differ from the internal models, stored in long term memory, that a programmer has of previously used programs.

3.4 Externalization

Though not many studies have been undertaken on this subject, it may be interesting to look at how programmers externalize programs, that is how they express their knowledge of a program.

On one hand, this may tell us what programmers really know about programs. For example, having some programmers use a program, and then asking them to recall it at different moments in time could show us which program features they really remember. These features would be the basic knowledge held in the long-term memory internal representations. Such experiments have been led by Davies, but they were based at short-term memory. They confirmed the theory of focal lines in programming plans.

On the other hand, it might be interesting to see how programmers spontaneously express their knowledge (either programs, problems or requirements), and which external representations they naturally use. As we said before, Petre (Petre, 1990) found that, when programming, programmers used a "private, pseudo-language". Other experiments of this type may point at which external representations may be the easiest to use for programmers.

The aim of all this would be to teach programmers what knowledge they should externalize in order to make their programs easily understandable by other people, and how.

4 Representations applied to reuse

The problem of studying the use of external memories and representations in software reuse is made more difficult by the fact that software reuse doesn't just consist in looking at programs, be they internal or external. Each kind of software reuse is made of different successive steps, which have their own aims and requirements.

In this section we will describe the four important steps of software reuse: finding a reusable asset, understanding it, specializing it and integrating it. We should keep in mind that these four steps are not completely independent, as some understanding occurs during the finding and specializing stages. This is just a simple model of the reuse activity.

We will see how these processes differ in the use of the internal representations, external representations, internalization process and externalization process. We will also see how they differ depending on the reuse technique used. Moreover, each step can be supported by some specific tools, and knowing which cognitive processes are involved in each step should help us designing those tools.

4.1 The four basic steps of software reuse

4.1.1 Finding a component

In one's internal memory In the case of a programmer trying to reuse one of his own programs (the code itself or the design), he must browse through his own long-term memory to find a suitable object to reuse. Then, he may remember different possible solutions, and still have to choose only one of them.

- *Searching one's memory*

  As we said before, we think that long-term memory can store various representations of existing programs. Though we don't know that much about those representations, we know that they can be of different abstraction levels. By 'browsing' through his memory, a programmer is able to recall different programs, and to judge wether they correspond to his problem. It seems natural to think that this search is a top-down process: from a general idea of the problem, or a few key concepts, the programmer can remember many possible solutions. The search can then be refined to reduce the number of solutions. This implies that the programmer is able to abstract his problem and requirements.

- *Choosing between alternative solutions*

  Once the programmer has found some possible solutions, he must choose only one of them. This is supposedly done by comparing their internal representations, and particularly how close they match the internal representation of the problem.

- *Solution evaluation*

  Finally, the programmer may evaluate how suitable the component is. Presumably, if it doesn't reach a minimum level of suitability, the idea of reusing may be discarded, and the programmer will write something from scratch.

Here we see that the situation may be different depending on wether the subject wants to reuse the actual code of a program, or just a design. For example, comparisons and choices between possible solutions for code reuse should ideally be based on code details like presentation or comments, which may not be present in abstract internal representations. Therefore, the judgements may be more efficient and reliable for design reuse than for source code reuse.

In an external memory The core of component reuse is to search through a library to find a suitable component. Here, library has a very wide meaning: it can be a pile of design or program documentation, a book of example programs or a computer-based database of code or design components.

- *Searching the library*

  'Browsing' consists in looking through a whole library, until something interesting is found. As such, this method is only suitable for small libraries, such as books of examples. As more complex library structures were developed (particularly database-like libraries), some automated search tools were developed to speed up searches. We will describe some of these in 4.2.1.

  Though different, all these tools require the programmer to externalize his internal representation of the problem. The important issue is to find an easy way for the programmer to express his problem. Reciprocally, since the programmer evaluates whether each component is suitable or not, he also has to internalize the description of the component. It seems natural to think that a simple description may be enough at first.

- *Choosing between alternative components*

  Once he has browsed or searched through the library, the user must choose one of the different possible components he obtained. Here the internalization process seems crucial, as the programmer will compare his internal representations of the possible solutions. Some more precise information on those components may be necessary.

- *Solution evaluation*

  Once a programmer has been through the whole process of searching a library and choosing one component, he can evaluate whether this component is suitable for his particular problem, that is, if the search was, as a whole, successful. If not, he may go on looking for a more suitable component (possibly in another library), or just give up and create something from scratch.

4.1.2 Program understanding

Once a programmer has chosen what he will reuse, he has to understand more thoroughly how this object works. In an idealistic situation, he wouldn't have to do this: by putting together well-crafted 'building blocks', a programmer would just need to know what each block does. But we know that this is never the case. We already had a look at the program understanding issue in 3.3, but here we can make some further remarks.

First of all, this step is crucial for external memory reuse, where external representations are involved. But it sometimes occurs for internal memory reuse. In most cases, the programmer remembers general aspects of the program or the design, which are sufficient for the reuse to take place. Yet he can sometimes remember details at a low level which he needs to abstract to really understand the component. For example, he may remember precisely the structure of the objects in an object-oriented program, and need to understand why this structure was chosen in order to reuse the same design.

Secondly, in the case of external memory reuse, a programmer may reuse a program he has written before, or somebody else's program. Reusing one of his own program will make him use his old internal representations as well as internalize the external representations (such as the code itself). It may be interesting to study when each kind of knowledge is used. This may show us what is missing from the internal representations of programs written in the past, and what knowledge a reuser really needs.

4.1.3 Specialization

Specialization is the process of adapting a general piece of code to one's own specific problem. The simpler the specialization process is, the more reusable the reused asset will be. This process depends on the reuse technique employed.

- *Write/Copy/Paste*

  When a programmer must write some variations of the same program, two kind of specialization processes occur. First, he uses his internal representation of the general solution, as well as the first instance's specific properties, to write the first instance. Then, he will use this first instance, which is an external representation, and the internal representation of the general solution, as well as each other instance's specifics, to build the other instances. Here we see that both internal and external representations, and therefore the internalization and externalization processes are involved.

- *Code scavenging*

  As Krueger (Krueger, 1992) put it, "a programmer specializes a scavenged code fragment by manually editing it". In order to know what must be specialized in the reused piece of code, the programmer presumably uses his internal representation of the program (which was built when he tried to understand it) and the internal model he has of his own problem.

- *Design scavenging*

  Here the programmer must specialize an existing design, supposedly using his internal representation of this design.

- *Code and design components*

  The use of ready-made components allowed for new specialization techniques to be developed. This specialization can be complemented by some manual editing.

  - Parameters

    A reusable component can accept parameters when it is reused, or more precisely invoked, such as the stack's maximum size for a stack component. The drawback of this method is that it is quite narrow, and its applications are limited.

  - Generic code

    A piece of code is said to be generic when it can use different types of input data: a programmer won't have to modify the component to take into account the data type he is currently using. The problem here is that it is quite difficult to accept a wide array of data types, since the programmer of the component always has to make some assumptions on the datatype he is using.

  - The white box

    The original view of software reuse was to be able to assemble software components without looking at how they achieve their task, i.e. using 'black-box' components. But this often seems too difficult to be possible. By 'opening' the components, by looking at how they do what we want them to do, we may be able to select a more suitable behaviour. This principle of the white box has been applied to the Open Implementation Analysis and Design methodology, developed by Maeda, Lee, Murphy and Kiczales (Chris Maeda & Kiczales, 1996).

4.1.4 Integration

When a reusable piece of code has been found and specialized, the programmer still needs to put it into his program. This can lead to some problems, such as name clashes, incompatibility, etc., which must be solved, according to Krueger (Krueger, 1992), by 'modify[ing] the fragment, the context, or both". Reused designs don't need any particular integration: they are high-level assets which ignore those low-level details. There are two different ways of integrating a piece of code in a program:

- *Code cloning*

For the Write/Copy/Paste technique, and in some cases for Code Scavenging, the piece of code needs to be fully inserted in one's program. This method may require many heavy modifications of the program. It will obviously involve the programmer's internal representation of the whole program and of the component. Though, once the modifications have been identified, the externalization process seems fairly simple.

- *Code invocation*

  By contrast, for other Code Scavenging situations and for component reuse, the component can simply be referred to, just like an external program, i.e. a C library or function. This method is much lighter and simpler, and helps ignoring low-level details. This allows the programmer to work on a higher level of abstraction.

4.2 Providing tools for library-based software reuse

Now that we have seen the basic steps of software reuse, it is interesting to look at how they can be supported by computer-based tools.

What we originally wanted to know was how the use of external memory disrupted the cognitive processes involved in software reuse. As we have just highlighted the cognitive issues of the four important steps of reuse, we can now see whether existing support tools are adequate from a cognitive point of view, and how some new tools might be developed from this perspective.

4.2.1 Finding a component in a library

Searching the library Most libraries now provide some kind of search tool. They first require the programmer to describe their requirements or problem, and give back a list of possible solutions. Search tools must tackle two cognitive issues: to help the programmer externalize his problem or requirements, and to help him selecting some possible solutions.

- *Externalizing the problem*

  First, we have to see how the programmer should express his problem or requirements. We can see three possibilities:

  - Browsing through a tree of choices will obviously make it very simple for the tool to understand the programmer's choices, but it will reduce the width of choice the programmer has (and thus make his description less accurate). As a matter of fact, the structure of the tree is most important. The tree's designer must choose which successive choices the user should make, and how components can be assigned to one or many set of choices. Such a system can be found in the AltaVista Web search tool, which can display an activation network in which the user can select a few nodes.
  - Alternatively, the system can ask for some keywords, which can be matched with each component's own list of keywords. This gives much more freedom to the programmer, as long as the tool can analyze the words and look for synonyms or equivalent words. Though, it requires the programmer to express himself just using a few words, making him choose what is really important in his problem or required solution.
  - Finally, it seems at first simpler to ask the user to express himself in natural language. His description of his problem or requirements can then be matched with each component's textual description. For example, some Web search tools can deal with queries such as "I want to know how to get from Brighton to London by train". The AltaVista search engine for example is very accurate on this particular request, while Excite doesn't give very useful links. But such a tool may be very difficult to develop in the case of component libraries, and the gain in freedom and simplicity from the programmer's point of view might be compensated or reversed by the inaccuracy or inefficiency of the tool. On top of that, a full textual description may contain useless information that can hide the most important features of the requirements. Keywords, by contrast, force the programmer to really think about what he wants. This is indeed an added cognitive weight on the programmer, but it may be at the end rewardful.

Besides, the programmer can describe two different things: a possible solution (which supposes that the programmer knows what he's looking for) or his problem (the tool then being able to match this problem to some solutions). Most keyword based tools allow the programmer to ignore this distinction, as keywords can indifferently describe the problem and the solution. By contrast, some natural language tools might provide a tool which will analyse the problem and find by itself what type of solution is required, though we still have to find how this can be done. The general idea is that it may well be simpler for a programmer, from a cognitive point of view, to express his problem rather than a possible solution, since problems are quite often already externalized, in the way of requirement drafts, contracts, etc.

As a conclusion, we can say that it may be interesting to conduct an experiment to support those ideas and see more thoroughly the advantages and shortcomings of the different methods of externalization.

- *Internalizing the component description*

  The other issue of searching in a database is to internalize the component descriptions, in order to find some components which, at first, might be suitable. Since this is just an early selection of components, it seems logical to think that a simple description of the component is sufficient. For example, the Asset Broker developed by BT gives short descriptions of just a few lines of its components, before giving access to longer descriptions. Most systems indeed use short textual descriptions (even sometimes the name of the component only) as an external representation. As a small amount of information is needed, textual description may well be sufficient.

  Besides, some search tools can provide added information which are adapted to the user's request. Yahoo! for example gives a probability rating (a percentage) of each Web page being about what the user is looking for. The user thus knows that Web pages rated at less than 70% might not be really interesting. The Asset Broker also gives an estimate of each component's suitability, using a Fair, Poor, etc. scale. This seems to be a very interesting trend to follow.

Choosing between different possibilities Here the internalization issue is crucial. The programmer has to 'cross-evaluate' a short-list of components. This undoubtedly requires further information on the components to allow a more precise evaluation and comparison. Though, the programmer may still not need to understand fully how the component works (for a code component) or its precise structure (for a design component). Yet the problem of which external representations to use remains. Most code or design component libraries provide textual descriptions at this stage. Some exceptions exist, such as the Design Patterns reference book (E. Gamma & Vlissides, 1995) (see 4.2.2) which also provides a small diagram of each pattern's structure.

This once again can be supported by some information the library provides about each component for this particular search, such as the Yahoo! probability rating. Though, as the crucial choice of which component will be reused is made at this stage, programmers may be reluctant to trust an automated tool, and tend to rely on their own judgement only.

Evaluating the chosen component Finally the programmer has to evaluate whether, as a whole, the chosen asset is suitable enough, or whether he should try to look in another library or give up and write the component himself. This has more to do with confidence in component libraries than cognition, though once again some tools can be helpful (such as the Yahoo! probability rating).

4.2.2 Understanding a component

The basic aim here is to provide an appropriate component description so that the reuser can understand what he needs to understand as easily as possible.

Existing descriptions and tools As far as source code or software architecture description is concerned, we saw in 3.2 different external representations, such as the control-flow, data-flow and structure (object hierarchy) diagrams. Though, as we said before, no particular representation has yet been proved to be in overall more efficient than the others. Instead, each one of them is more suitable for some particular tasks.

Design patterns have been described by Gamma, Helm, Johnson and Vlissides in (E. Gamma & Vlissides, 1995), by means of a 12-point list:

- *Intent* The short description of the purpose of this pattern
- *Also known as* An older name
- *Motivation* An abstract example of a situation where this pattern is useful
- *Applicability* Classic situations where we can use this pattern
- *Structure* The pattern itself
- *Participants* Other patterns that this pattern uses
- *Collaboration* Other patterns' behaviour re this pattern
- *Consequences* The advantages and shortcomings of the pattern
- *Implementation* Some techniques and advices to implement the pattern
- *Sample code* A complete example
- *Known uses* Actual uses in available programs
- *Related patterns*

The *Intent* might be used as a short description, for the early selection of a few possible patterns. A small diagram of the pattern's structure is provided as well.

Further developments Here we can try to see which directions could be followed to properly estimate which external representations are the most effective, or to try to design some new external representations.

- *Externalizing internal representations*

  A possible solution may be to map external program representations on our internal representations. The problem is that, as we said before, we do not know a lot on how our internal representations of programs are structured and organized.

  Object-oriented programming for example was such an attempt, since (as Ormsby (Ormsby, 1996) recalls it), it was usually claimed that OO programming (like nearly every new programming language in fact) 'introduces a software development model based upon the way humans think'. We could say for instance that the 'object' concept can be linked to the 'node' concept in the semantic network model of human memory. Inheritance and method calls could then be seen as the external equivalent to the connections between concepts. This was claimed by Goldberg and Robson (Goldberg & Robson, 1986) about the Smalltalk 80 object-oriented language.

  Yet this claim still has to be proved, and it may even be wrong as, for example, Lee and Pennington (Lee & Pennington, 1994) showed that it was usually more difficult to understand an object-oriented program than a procedural one, since domain knowledge (as opposed to the task itself) tends to be more important in object oriented programming than in functional programming.

- *Natural representations*

  Another solution may be to look at which external representations programmers naturally use when they program, for example by looking at the small drawings they make while programming, and the notes they take. This may hint at which external representations programmers feel confident with and understand easily. This might give us some clues on what 'good' external representations should look like, though Petre showed that these representations differ from one individual to another.

- *Human laziness*

  Making programmers understand some piece of code that they are reusing may be more difficult than we think. Lange and Moher (Lange & Moher, 1989) described the "avoidance of comprehension" phenomenon: programmers are not inclined to make the (maybe unnecessary) effort to fully understand what they reuse. This has also been highlighted by Sutcliffe and Maiden (Sutcliffe & Maiden, 1990), who said that programmers are usually "copying rather than reasoning while reusing specification components". The

consequence is that providing a complete and accurate description of software components is probably not the ideal solution. Instead, by motivating reusers to actually try to understand the components they reuse, and more importantly by identifying the knowledge they really use in the program descriptions (that is, the information that they really need), we may find a way to make external representations shorter, simpler, and more efficient.

- *Programmers are individuals*

  Another problem is that Davies, Gilmore and Green (S. P. Davies & Green,) showed that programmers do not classify pieces of code on the same criteria. Novice programmers tend to use surface criteria (such as class names), whereas expert programmers prefer deep semantic aspects, which may not seem obvious at first (such as some aspects of the code itself). Thus maybe different types of users also need different types of information to reuse a component. A solution to this problem may be to setup a library system that can alternatively display different kinds of representations, or allow the user to parameterize these representations, or even better that can build a profile of each user and display his favored representations.

4.2.3 Specializing a component

As we said before, specialization is the process of adapting a general piece of code to one's precise problem.

This step is quite different for Design Patterns. DPs don't need to be really specialized nor integrated into a program: instead they must be assembled together like building blocks. This was the original idealistic view of software reuse. From this point of view, DPs are the most successful kind of software reuse. Yet to assemble them is still quite like a programming task, but at a higher level, and combinations of patterns can be reused. It may be interesting anyway to find how a computer-based tool might support this assembly process.

As far as code components and software architectures are concerned, developments in new specialization techniques went in two opposite directions. The 'black-box' concept consists in hiding from the user what the component actually does (by restricting the specialization to some parameters or data types), while the 'white-box' concept tries to 'open' the component and automate the specialization at a low-level (in the case of the Open Implementation Analysis and Design methodology for example).

Apparently, the situation for code reuse is quite problematic: black-box components are not reusable enough, and white-box components are too difficult to specialize. Unless some more efficient and easy-to-use tools are developed, it seems logical to think that design-level reuse will focus most of the interest in the future.

4.2.4 Integrating a component

Finally the programmer must solve the disruption caused by integrating the new component in his program. This step does not usually occur for Design Patterns, unless only a part of a larger system has been redesigned with DPs. We can think that it is also not relevant for Software Architectures reuse where (a) a larger, stand-alone program is reused, and (b) this is done at a high level of abstraction, where problems such as name clashes do not exist.

As far as code reuse is concerned, we saw that the original technique was to copy/paste the component into the program, and that languages now supported the use of function calls to simplify this. That way a program is indeed more like a set of modules than one single list of commands. This system seems to be a satisfactory solution, as the frontier between the program and the component has been reduced to something which is easily understandable and manageable by programmers.

5 Conclusion

We have described different reuse techniques, and how the most interesting ones are based on the use of external memories. We also explained how the reuse process could be split in four steps. Though these steps are not completely independent (for example, some understanding occurs during the search and specialization stages), some computer-based tools can support each one of them.

Taking into account cognitive aspects of software reuse, we tried to bring new ideas about the design of these tools. But the main problem is that we still do not seem to know enough about some key issues such as the internal representations of programs. Therefore we could only draw up a list of advice, suggestions and ideas.

What appears is that on many issues, some alternative solutions have been suggested, without them being supported by any comparative experiments. Therefore we think that it would be interesting to develop a modular

reuse system which would support such experiments. This system would be based on a set of modules, where each module can be chosen from a few alternative possibilities. These modules might include:

- the component type: code, design architecture, or design pattern
- what the user should externalize for the search: the problem or the behaviour
- how he should externalize it: using keywords, natural language, or by browsing through a tree of choices
- what knowledge should be displayed about each component for the early selection stage
- how this knowledge should be displayed: text, graphics, data or control flow diagrams, a combination of them, etc.
- what knowledge should be displayed, and how, for the cross-evaluation stage
- a specialization tool: to display further information, or to automate this process
- an integration tool

It would then be possible to ask some people to perform some kind of reuse activity with a given module configuration, and then with a slightly different one, and to draw comparisons between the efficiency of those configurations. It would also allow us to see whether different people prefer different configurations, and to identify some user profiles (for example, depending on the user's experience). Finally we would be able to isolate which kind of information users really need, by providing more or less complete representation components, and by comparing them.

Such a tool should not be designed towards supporting reuse in itself, but rather as a simple and modular system which will support experiments about software reuse.

References

Brooks, R. (1983). Towards a theory of the comprehension of the computer programs. *International Journal of Man-Machine Studies*, pp. 543–554.

C. M. Hoadley, M. C. Linn, L. M. M., & Clancy, M. J. (1995). When, why and how do novice programmers reuse code?. http://obelisk.berkeley.edu /tophe/ESP6/ESP95Final.html.

Chris Maeda, Arthur Lee, G. M., & Kiczales, G. (1996). Open implementation analysis and design. http://www.parc.xerox.com/spl/projects/oi-at-parc/ourpapers/oiad.pdf.

Davies, S. P. (1995). Focal structures in program comprehension: Implications for the design of programming support tools, debugging aids and tutorial environments. In Y. Anzai, K. O., & Mori, H. (Eds.), *Symbiosis of Human and Artifact*. Elsevier Science B.V.

Détienne, F. Reasoning from a schema and from an analog in software code reuse. In *ESP-4*, pp. 5–22.

E. Gamma, R. Helm, R. J., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Reading, MA, USA.

Goldberg, A., & Robson, D. (1986). *Smalltalk 80 - The language and its implementation*. Addison-Wesley.

Green, T. R. G. (1989). Cognitive dimensions of notations. In Sutcliffe, A., & Macaulay, L. (Eds.), *People and Computers*, Vol. 5. Cambridge University Press, Cambridge, UK.

Green, T. R. G. (1990). Programming languages as information structures. In J.-M. Hoc, T. R. G. Green, R. S., & Gilmore, D. J. (Eds.), *Psychology of Programming*, pp. 117–137. Academic Press - Harcourt Brace Jovanovich, London, UK.

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys*, *24*(2).

Lange, B. M., & Moher, T. G. (1989). Some strategies of reuse in an object-oriented environment. In Bice, K., & Lewis, C. (Eds.), *Proceedings of CHI'89* New-York, USA. ACM Press.

Lee, A., & Pennington, N. (1994). The effects of paradigm on cognitive activities in design. *The International Journal of Human-Computer Studies*, pp. 577–601.

McIlroy, M. D. (1968). Mass produced software components. In Naur, P., & Randell, B. (Eds.), *Software Engineering: Report on a Conference by the NATO Science Committee (Garmisch, Germany, Oct.)*, pp. 138–150. NATO Scientific Affairs Division, Brussels, Belgium.

Neighbors, J. M. (1984). The draco approach to constructing software from reusable components. *IEEE Trans. Soft. Eng.*, *10*(5), 564–574.

Neighbors, J. M. (1989). Draco: a method for engineering software systems. In Biggerstaff, T. J., & Perlis, A. J. (Eds.), *Frontier Series: Software Reusability: Volume 1 - Concepts and Models*, pp. 295–319. ACM Press, New-York, USA.

Ormerod, T. (1990). Human cognition and programming. In J.-M. Hoc, T. R. G. Green, R. S., & Gilmore, D. J. (Eds.), *Psychology of Programming*, pp. 63–82. Academic Press - Harcourt Brace Jovanovich, London, UK.

Ormsby, A. (1996). Software engineering and cognitive science. School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, pp. 295–341.

Petre, M. (1990). Expert programmers and programming languages. In J.-M. Hoc, T. R. G. Green, R. S., & Gilmore, D. J. (Eds.), *Psychology of Programming*, pp. 103–115. Academic Press - Harcourt Brace Jovanovich, London, UK.

Prieto-Diaz, R., & Freeman, P. (1987). Classifying software for reusability..

S. P. Davies, D. J. G., & Green, T. R. G. Are objects that important? the effects of expertise and familiarity on the classification of object-oriented code..

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *10*(5).

Sutcliffe, A., & Maiden, N. (1990). Software reusability: Delivering production gains or short cuts. In D. Diaper, D. Gilmore, G. C., & Shackel, B. (Eds.), *Human-Computer Interaction - INTERACT'90* Amsterdam, Netherlands. Elsevier.

von Mayrhauser, A., & Vans, A. M. (1995). Industrial experience with an integrated code comprehension model..

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

Intention movements and the evolution of animal signals

Jason Noble
jasonn@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract Two animals contesting possession of a resource sometimes fight, but more often engage in aggressive displays until one or the other retreats. Ethologists such as Tinbergen and Lorenz suggested that such threat displays were honest signals of aggressive intent, and evolved through the ritualization of "intention movements"—movements that reliably predict subsequent behaviour. More recent thinking in theoretical biology has shifted to a gene-centred view where signalling is seen as the manipulation of one animal by another, but there is still room for the concept of intention movements as evolutionary seeds. A simulation, inspired by Maynard Smith's Hawk–Dove game, is presented in which two agents fight for a resource and have access to the ongoing strategic choices of their opponent. The results from the simulation are used to cast light on conflicting theories of the evolution of communication, and the conditions under which stable, honest signalling systems should be expected.

1 Introduction

This paper describes a simulation model of the evolution of aggressive signalling in animal contests. Animals contesting the possession of a resource such as food or a mating opportunity sometimes fight, but more often they engage in threat displays—a mutual show of strength—until one animal (typically the weaker one) retreats, leaving the other with the resource. Examples from nature include stags fighting over harems, vultures fighting over the best bits of a carcass, etc.

Intuitively, settling contests by signalling makes sense. We can see that an all-out fight is usually a bad idea: fighting is energetically expensive, and there is always a risk of injury or death. So how might the tendency to settle disputes through signalling have evolved?

The early ethologists, notably Niko Tinbergen and Konrad Lorenz, suggested that animal threat displays could evolve through the gradual ritualization of "intention movements", i.e., movements that reliably predict subsequent behaviour. A bird cannot fly without first spreading its wings; a dog cannot bite without first baring its teeth—wing-spreading and teeth-baring are thus intention movements, and ethologists suggested that these were candidate raw materials for the evolution of signals. The evolutionary story looks pretty obvious: dogs have to bare their teeth before biting, a mutant dog comes along that picks up on this and retreats when other dogs bare their teeth at it, the mutant does well because it avoids costly fights, teeth-baring is subsequently exaggerated for clarity and becomes an effective threat display.

Tinbergen and Lorenz were great observers of animal behaviour, but the problem with some of their ideas is that they incorporate the group-selectionist thinking that was then prevalent in biology. They suggested that intention movements would give rise to *honest* signals of strength, or perhaps of "willingness to fight", because this would help the species avoid unnecessary and costly conflicts.

Unfortunately, the idea that animals do things for the benefit of the species was almost entirely discredited in the modern move towards gene-centred thinking in evolutionary biology. Biologists now believe that selfishness is the norm (Williams, 1966; Dawkins, 1989). In terms of signalling, the logic has become that of manipulation: Dawkins and Krebs (1978), and Krebs and Dawkins (1984), argued that signals are the way one animal manipulates another into doing something beneficial to the signaller. Under this view, intention movements might still provide raw material for the evolution of signals, but the explanatory story is quite different: dogs have to bare their teeth

	. . . plays Hawk	. . . plays Dove
Hawk	-1	2
Dove	0	1

Table 8: Payoff matrix for the Hawk–Dove game

before biting, a mutant dog picks up on this and retreats when other dogs bare their teeth, and for a while the mutant does well because it avoids costly fights. However, once this "mind-reading" variant has become common in the population, the stage is now set for the arrival of a second mutant: a dog that bares its teeth even when it is weak or has no intention of fighting. The second mutant exploits the behaviour pattern of the first in order to *manipulate* animals that may be much stronger into walking away from the contest.

As you might guess, such a situation is not going to be stable. Dawkins and Krebs predict a cycle of increasingly inflated signals (e.g., teeth bared more prominently and for longer) and increasingly sceptical, "sales-resistant" receivers. The ethologists also thought that signals would be exaggerated, but in the interests of reducing ambiguity. In contrast, Dawkins and Krebs argue that it is rarely in an animal's interest to reduce ambiguity about its strength or its intention to fight.

Game-theoretic models have gone hand in hand with the gene-centred approach in biology. Game theory is a useful way of looking at problems in behaviour such as signalling, where the effectiveness of any one strategy depends on what other members of the population are doing. Maynard Smith (1982) developed the classic Hawk–Dove game (see table 8) to look at what happens when animals can either signal or fight over a resource. In the game, animals adopt one of two strategies: Hawk or Dove. Hawks always fight until they win or until they are seriously injured, whereas Doves try to settle the contest by signalling, and will retreat if attacked. It is assumed that when two Hawks meet, one will gain the resource and the other will be injured. When two Doves meet, one "wins" the signalling duel and gains the resource, and the other retreats without being injured. When a Dove meets a Hawk, the Dove retreats immediately and the Hawk gets the resource. Maynard Smith demonstrated that if the resource is worth more than the cost of injury, then the only *evolutionarily stable strategy* (strategy invulnerable to invasion) is to play Hawk all the time. However, when being injured costs more than the resource is worth, things get interesting: it turns out that the only stable strategy is a mixture of Hawk and Dove, realized either as a stable polymorphism of pure Hawks and pure Doves, or as individuals who sometimes play Hawk and sometimes Dove.

If the contested resource is worth 2 units, and the cost of being seriously injured is 4 units, the stable strategy is a 50–50 mix of Hawk and Dove, and the payoff matrix works out as shown in table 8. The reason a Hawk playing a Hawk has an expected payoff of -1 is because they will win the resource (2 points) half the time, and suffer injury (-4 points) half the time. In the simulation described here, things cannot be reduced to a simple payoff matrix, but nevertheless the cost functions have been designed so that a similar situation obtains—on the face of it, the resource is not so valuable that it is worth risking injury.

A key point to notice, in the Hawk–Dove payoff matrix, is that if you *knew* your opponent was going to play Hawk, you would do better by playing Dove. And vice versa. The question that my simulation tries to get at is this: what happens if animals are playing out their encounter in real time instead of the discrete one-off decision of game theory, and have access to information about the ongoing strategic choices of the other, i.e., if they could watch the other wavering between choosing to play Hawk and choosing to play Dove?

The subject of information exchange during animal contests is far from straightforward. Game-theoretic models have been used to investigate a range of cases more complex than the basic Hawk–Dove game, in which animals can differ in their fighting ability, in the relative value of the resource (e.g., one is hungrier than the other), and in other ways that might serve to break symmetry in contests, such as one animal being the current owner of the resource and the other being an intruder. The consensus from these models is that it can be stable for contests to be settled by any of these asymmetries (in preference to fighting) as long as information about the asymmetry is available to both animals, and cannot be faked. So, for example, the roaring contests and parallel walks of red deer stags (Clutton-Brock, Albon, Gibson, & Guinness, 1979) are explained as assessment signals: the stags are exchanging unfakeable information about relative strength, and the contest will be settled without any actual fighting unless the two animals are very closely matched. If there is an asymmetry, i.e., if one stag is stronger than the other, then the contest will be decided accordingly.

But what about cases where information about asymmetries is not out in the open? For example, if you are playing the Hawk–Dove game, and you intend to play Hawk no matter what, then information about that intention is presumably only available to you. Paradoxically, even though intention movements might be the raw material for signals, game-theoretic models predict that it will not be stable in the long run to transmit information about intentions. In this respect, Maynard Smith is fond of the poker metaphor: whether you have four aces or a pair of twos you should not say so; and if your opponent claims to have a good hand you should not believe him.

The poker metaphor is compelling, and the situation modelled here is similar: animals have access to information about their own fighting ability, but not their opponent's ability. Like poker players, animals would of course *like* to know their opponent's strength, because then they could make better strategic choices and avoid fights that they could not win. The question is, if these animals can each perceive the other's intention movements, will they evolve signals that give away their own fighting ability, or will they remain "poker-faced", and conceal or exaggerate their true ability?

Game-theoretic models start to become intractable when we consider the possibility that animals have access to information about their own strength, and can potentially exchange that information with their opponent in a sustained interaction. However, a recent model by Hurd (1997) suggests that if two competing animals can send one of two possible signals before deciding whether to fight or flee, it is in fact stable for them to use this signal as an honest indicator of fighting ability. Hurd's result is a little surprising because he found that even when the signal cost nothing to send, honesty was stable, and that if one of the two alternative signals had a cost attached, it was only stable for *weaker* animals to use the costly signal. Further, if cost-free signals were available, they would be used in preference to costly ones. Hurd's result is surprising becomes it stands in the context of a wider controversy in the signalling literature as to whether we should expect animal signals to be honest or not, and, if and when they are honest, what costs serve to maintain their honesty, etc (Zahavi, 1975, 1987; Grafen, 1990). Recently the current of biological thought had swung around to the idea that honest signals had to be inherently costly, like peacock tails and deer roaring.

2 Method

2.1 The contest

The model assumes that the essential quality of an intention movement that predicts, for example, attack is that you have to display the movement (physiologically "have to") before attacking, but you could make the movement and then not attack. So, one could reach for a cup and then not pick it up, but could never pick it up without first reaching for it.

The two movements of interest were attacking the opponent, and fleeing from the contest, i.e., abandoning the resource. To keep things simple, there is a single behavioural continuum between these two options: the simulated animals start at a neutral value, to move "up" is to attack, and to move "down" is to retreat.

Attacking is defined as crossing a threshold a certain distance above the neutral value; fleeing from the contest is crossing a negative threshold. To visualize this, imagine that the horizontal axis is time (show figure 1).

As soon as one animal crosses the "flee" line, the contest is over. For every time-step that an animal remains above the "attack" line, it is assumed to be causing injury to its opponent, and if this goes on for long enough, the opponent will be physically overcome. Animals are of either high or low fighting ability; high fighting ability means being able to cause more damage per time-step. The intention-movement aspect consists of the fact that animals cannot jump straight to "flee" or "attack" within a single time-step, but must move there gradually.

2.1.1 Contest resolution

The contest can end in one of three ways:

1. one animal fleeing
2. one animal overcoming the other in a fight
3. time running out (max. 750 time-steps)

2.1.2 Costs and benefits

Costs are of two types:

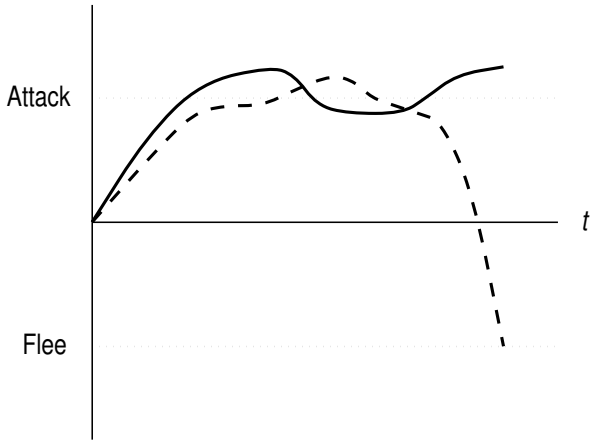

Figure 1: Winning a contest through a combination of fighting and threatening the opponent. The weaker animal (dashed line) flees after the stronger animal shows a greater willingness to back up "threat displays" by attacking.

1. -0.4 units (scaled) per time-step for aggression
2. -0.8 / -3.2 units per time-step for being attacked by a weak / strong opponent

Once an animal reaches a total of -340 units *for that particular contest* it is assumed to be have been physically overcome or exhausted and it loses the resource to its opponent. The only benefit in the game is to gain the resource: +200 units.

Note that there is a modest cost involved in aggression, i.e., anything that happens above the neutral axis. This seemed reasonable, in that attacking an opponent should presumably cost something in energy expenditure, and thus intention movements toward attacking might also be expensive.

2.1.3 Inputs and outputs

The animals "know":

1. Their own strength (randomly determined for each contest)
2. Their own position on the Attack–Flee continuum
3. Their opponent's position on the continuum

The animals also have access to a random input, to allow for stochastic strategies (e.g., being Hawk-like half the time and Dove-like the other half). The only output the animals produce is a distance to move up or down on the Attack–Flee axis.

2.2 Predictions

Given Hurd's result, it was hypothesized that communication would occur, but possibly with occasional incursions of bluffing. I also thought that if the simulated animals did communicate, they would reach an equilibrium where no fighting occurred between Strong–Weak pairs. I suspected that in cases of asymmetry something like figure 2 would occur.

Because there is a cost involved in any activity above the neutral level, and because Hurd had suggested that cost-free signals would be used to exchange information when available, it seemed possible that signalling would occur *below* the neutral axis, where all activity is cost-free.

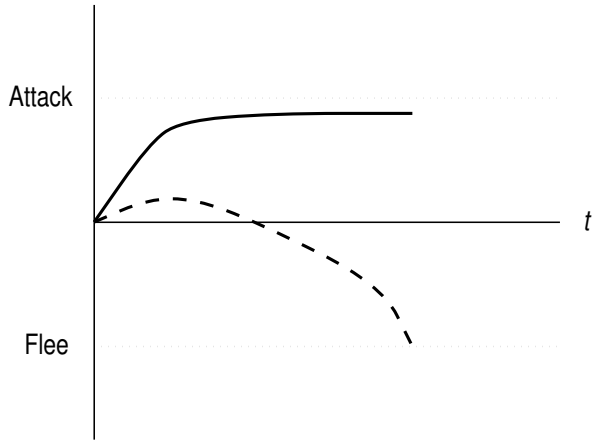

Figure 2: Winning a contest by threatening the opponent. The weaker animal (dashed line) abandons the resource and flees after a threat display by the stronger animal. Note that neither animal has actually attacked the other.

2.3 Behind the scenes

The animals were implemented as five-neuron fully inter-connected continuous-time recurrent neural nets, with all parameter values taken from Yamauchi and Beer (1994). This architecture was treated as a black box and I have assumed that it is capable of producing an approximation of any conceivable strategy; I have not tried to analyze the internal dynamics of the nets.

The genetic engine was a standard GA with a population size of 100, run for 5000 generations, using roulette-wheel selection. In each generation, animals were randomly selected to play out 500 contests; each animal could thus expect to play 10 games per generation.

2.4 Control groups

In order to tease out the patterns of causation, two control groups were devised.

Zero-information control: animals have no access to the position of their opponent on the Attack–Flee continuum. There is thus no possibility for communication and the animals must choose a strategy based only on their own strength.

Full-information control: animals are permitted to know their opponent's strength. In a sense there will be no communication here either, because the main object of such communication has been served up for free. When animals are equally matched, perhaps they will pay attention to each other's intention movements, but when they are mismatched in strength they will know it immediately and presumably act accordingly.

3 Results

In simulation models like the one presented here, it is statistically desirable to present results that are averaged over a number of runs starting with different random seeds. However, the current model stands as a pilot study for future, more detailed investigations, and thus the results will refer only to a single, typical run in each experimental condition. The raw data, as is typical in these sorts of simulations, were extremely spiky, and all graphs have been smoothed over a 50-generation moving average.

3.1 Fitness

Given the co-evolutionary nature of the simulations (i.e., the same population have to become good signallers and good receivers, if you like) the plot of mean fitness over time did not march steadily upwards but was fairly stable.

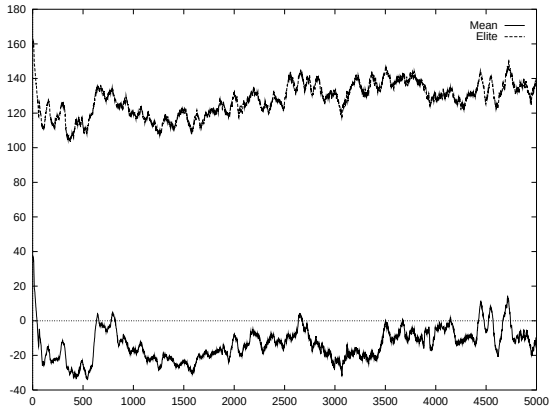

Figure 3: Mean and elite fitness per generation in the experimental condition.

Table 9 shows the mean fitness across the three conditions.

	Mean fitness	Standard deviation
Experimental condition	-13.9	20.2
Zero-information control	-10.7	15.7
Full-information control	44.7	14.8

Table 9: Mean fitness (expressed as mean payoff per individual per game) across the three experimental groups over 5000 generations.

The first thing to note is that supplying the full-information control group with knowledge of their opponent's strength is clearly valuable to them: this group has a much higher mean fitness. However, it is not so clear that supplying knowledge or perception of the opponent's intention movements, as happens in the experimental group, is of value. The experimental group actually does slightly *worse* than the zero-information control, in which the animals have no knowledge whatsoever of their opponent's actions. At first glance, it seems as though awareness of the intention movements of others, with all the potential for bluff and double-bluff that that entails, has just made things worse.

3.2 Behavioural strategies

Still, what are the simulated animals actually doing? We can see in figure 4 that most of the time the contests are being settled by one animal running away rather than an all-out fight, and also that running out of time is relatively uncommon.

3.3 How to measure communication?

The observation of different behaviour patterns across the experimental conditions are interesting and probably warrant further analysis. However, what we need is a simple measure that will tell us conclusively whether any communication is going on.

From observing the progress of the runs on a graphical display, I realized that the predictions of figure 2 were not entirely borne out. Weaker animals did often move in an arc towards aggression and then back towards fleeing once they "saw they were outmatched". However, no horizontal threat displays were observed; strong animals in particular tended to advance rapidly to the attack threshold and then just keep going to very high values. (This was interesting, as there was no apparent advantage to going far above the threshold: once you were over the line, you were attacking the opponent.)

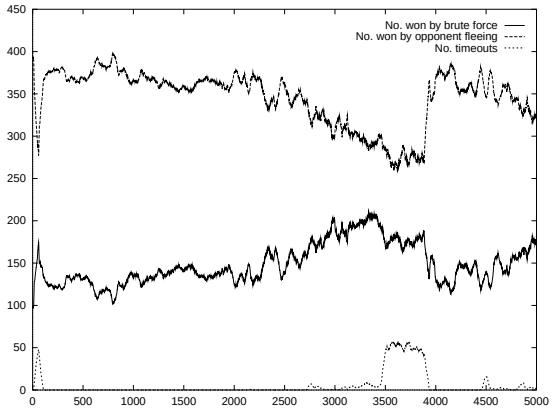

Figure 4: Overall behavioural outcomes in the experimental condition.

Given that only the animal "knew" its own strength, it seemed the best way to measure communication was to look at whether the behaviour of the *other* animal was different for animals of different strength. For instance, do weak animals behave differently against strong and weak opponents? In one sense (see section 3.2) they obviously do—even in the zero-information control, weak animals are much more likely to chase away a weak opponent than a strong one. However, this sort of difference is a function of overall strategy based on the animal's own strength. In the zero-information control, strong animals move up, and weak animals move hesitantly down. If you are a weak animal facing a weak opponent, sometimes you will reach the flee line first, and sometimes your opponent will. That does not mean that you have acted differently based on your opponent's strength.

Figure 5 shows a plot of the mean positions of strong and weak animals, playing strong and weak opponents, at time-step 8 in the contest. Time-step 8 is quite early in the piece, and almost all the bouts are still continuing at this point, except for a few where one animal has fled extremely quickly. It is much too early for either animal to have overcome the other by fighting. Note that 100, on the y-axis, is the attack line.

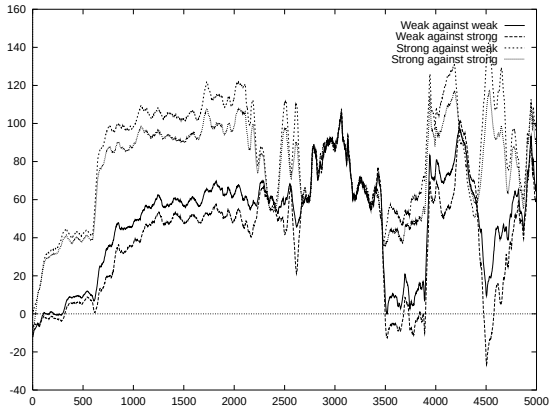

Figure 5: Mean positions on the Attack–Flee axis at time-step 8 in the experimental condition, for weak and strong animals against weak and strong opponents. Note that 100 on the vertical axis represents the attack threshold.

Notice that strong animals are generally showing a higher level of aggression than weak animals at this point, typically attacking or almost so. That makes sense. However, after a few hundred generations, the animals themselves seem to have been able to make use of the differing stereotypical responses for strong and weak, and both classes of animal are now, on average, a little more aggressive against weak opponents. My claim is

that this constitutes communication—for the strength of animal A to be influencing the behaviour of animal B, information must have been transmitted. As an analogy, imagine telling one person "high" or "low", then asking another to guess what you had told the first, and then repeating the exercise many times. If the second person was guessing better than chance what you had told the first, then we would suspect that some sort of communication was occurring between them.

The animals are signalling strength or weakness by the particular gradient at which they move towards attacking behaviour. Notice that there seems to be an inflationary aspect to the signal: what signifies strength at generation 500 is later typical of weakness. And of course, the communication seems to be unstable.

Figure 6 is a plot of the "communication indices", defined as the difference between responses at time-step 8 for each class of player to strong and weak opponents. It is clear that communication is established, fluctuates, then disappears and is replaced by a period of poker-facedness around generation 3000, where both classes of animal are behaving in exactly the same way. It is re-established later on.

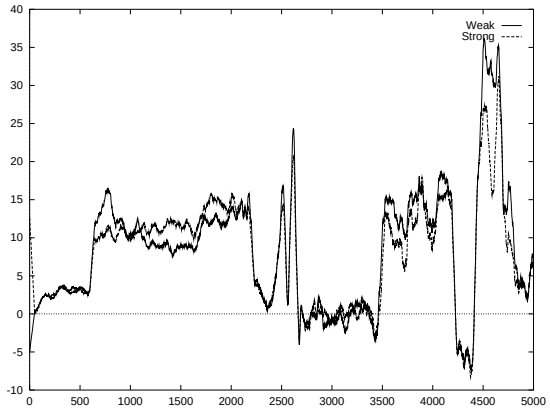

Figure 6: Communication indices in the experimental condition. The communication index for strong and for weak animals respectively is the mean distance by which they were more aggressive towards weak than strong opponents at time-step 8.

If these communication indices are really a good measure of the transmission of information regarding fighting ability, then we would predict that they would be constantly zero in the zero-information control, and consistently high in the full-information control. These predictions are borne out in figures 7 and 8 respectively.

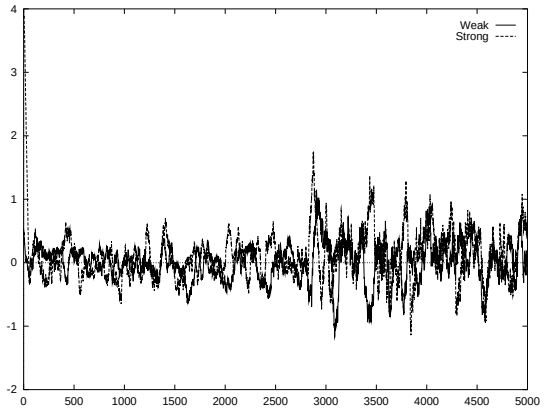

Figure 7: Communication indices in the zero-information condition.

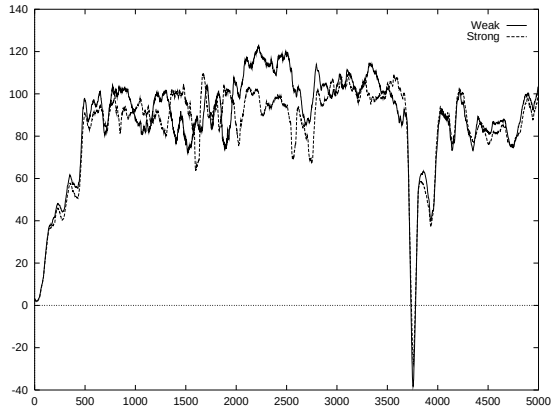

Figure 8: Communication indices in the full-information condition.

4 Conclusions

The simulation shows that communication can evolve, albeit unstably, between agents who know their own strength but not that of their opponent and can perceive each other's intention movements in a contest. This finding is worth feeding back into the debate in theoretical biology about when we should expect honest signalling to occur. Hurd's model is partly validated, in that there was at least *some* honest signalling of fighting ability, but there was no tendency for the simulated animals to use the cheap signalling zone available in the bottom half of the graph. I suspect that the biggest difference between Hurd's model and my own is that for Hurd the signals are pre-supplied, arbitrary, and divorced from any other function. In most A-Life work on communication (Werner & Dyer, 1991; MacLennan & Burghardt, 1994) things are a little more detailed but there is still a dedicated, artificial communication channel given *deus ex machina* to the simulated animals. In nature it is obviously not like that, and I hope that this model goes some way towards helping us to understand what must happen when evolution co-opts an existing behaviour for the new purpose of signalling.

4.1 Thoughts for future variations

Continuous variation in fighting ability would be a good thing to explore next. Maynard Smith (1994) suggests that cost-free signalling is easier to get when the model assumes discrete distribution of fighting ability, and may be impossible in truly continuous distributions.

There are several issues about whether the architecture and informational inputs that I have given to the simulated animals are in fact adequate for them to develop any conceivable strategy that selection might push them towards. Given that there has been absolutely no effort to make the control architecture biologically plausible, it would be disastrous if artefacts related to that architecture were in fact limiting the behaviour patterns of the animals. Therefore in future work I plan to give the animals a time input, in case the absence of this information is preventing the development of "horizontal" signalling. Along similar lines, I want to use twin neural nets for "strong" and "weak" strategies, just in case too much is being asked of the single existing net.

Also, I would like to do a more basic simulation where it was not necessarily assumed that fighting was a more basic ability that had evolved prior to the ability to make sense of an opponent's intentions.

References

Clutton-Brock, T., Albon, S. D., Gibson, R. M., & Guinness, F. E. (1979). The logical stag: adaptive aspects of fighting in red deer *(Cervus elephos L.). Animal Behaviour*, *27*, 211–225.

Dawkins, R. (1989). *The Selfish Gene* (Revised edition). Oxford University Press, Oxford.

Dawkins, R., & Krebs, J. R. (1978). Animal signals: Information or manipulation. In Krebs, J. R., & Davies, N. B. (Eds.), *Behavioural Ecology: An Evolutionary Approach*, pp. 282–309. Blackwell, Oxford.

Grafen, A. (1990). Biological signals as handicaps. *Journal of Theoretical Biology*, *144*, 517–546.

Hurd, P. L. (1997). Is signalling of fighting ability costlier for weaker individuals?. *Journal of Theoretical Biology*, *184*, 83–88.

Krebs, J. R., & Dawkins, R. (1984). Animal signals: Mind reading and manipulation. In Krebs, J. R., & Davies, N. B. (Eds.), *Behavioural Ecology: An Evolutionary Approach* (Second edition)., pp. 380–402. Blackwell, Oxford.

MacLennan, B. J., & Burghardt, G. M. (1994). Synthetic ethology and the evolution of cooperative communication. *Adaptive Behavior*, *2*(2), 161–188.

Maynard Smith, J. (1982). *Evolution and the Theory of Games*. Cambridge University Press, Cambridge.

Maynard Smith, J. (1994). Must reliable signals always be costly?. *Animal Behaviour*, *47*, 1115–1120.

Werner, G. M., & Dyer, M. G. (1991). Evolution of communication in artificial organisms. In Langton, C. G., Taylor, C., Farmer, J. D., & Rasmussen, S. (Eds.), *Artificial Life II*, Vol. X of *SFI Studies in the Sciences of Complexity* Santa Fe Institute, NM. Addison-Wesley, Redwood City, CA.

Williams, G. C. (1966). *Adaptation and Natural Selection*. Princeton University Press, Princeton, NJ.

Yamauchi, B. M., & Beer, R. D. (1994). Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, *2*(3), 219–246.

Zahavi, A. (1975). Mate selection—a selection for a handicap. *Journal of Theoretical Biology*, *53*, 205–214.

Zahavi, A. (1987). The theory of signal selection and some of its implications. In Delfino, V. P. (Ed.), *International Symposium on Biological Evolution*, pp. 305–327.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

Teaching a Learning Companion

Jorge A. Ramírez Uresti *
jorgeru@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract Learning Companion Systems (LCS) are a variation of Intelligent Tutoring Systems (ITS). In an LCS, besides the traditional tutor and student, a new agent is introduced: the Learning Companion (LC). The issues of the expertise and behaviour that such a companion agent should have, if it is going to be of any use to the student, are very important in these systems. This paper explores the hypothesis that a less capable learning companion is helpful to a human student by encouraging her to teach the LC.

Introduction

An Intelligent Tutoring System (ITS) can be seen as a system with two agents: a tutor and a student (figure 1). A criticism of such systems is that they are inherently based on one-to-one interactions between a student and a tutor and cannot encompass the richer learning possibilities opened up by involving more than one learner.

Learning Companion Systems (LCS) were first introduced by Chan and Baskin (1988) influenced by the idea of collaborative partners (Gilmore & Self, 1988; Cumming & Self, 1989). These are systems which attempt to model groups of learners as in a classroom. An LCS consists of at least three agents. The tutor and the student remain the same as in an ITS. The new addition is an agent called the Learning Companion (LC), or just the companion (figure 1). The role of this new agent is to be a peer of the human student. In principle this companion should be helpful to the student in a number of ways. For example, the companion could be a role model for the student; both students could collaborate and compete as equals (Chan & Baskin, 1990); the companion could be a source of advice (Hietala & Niemirepo, 1996); the companion could be a student of the human student.

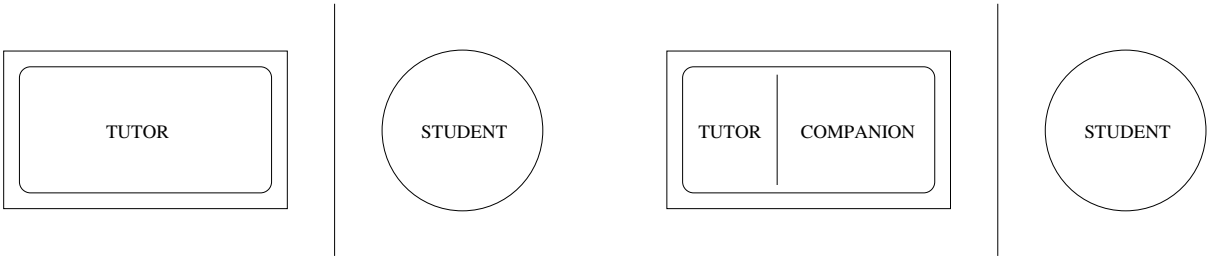

Figure 1: ITS vs. LCS

This paper describes work in this final category. A LCS is being built to explore the hypothesis that a less capable learning companion is helpful to a human student when learning. The student will be encouraged to learn by teaching this kind of companion. The question of how to motivate the student to interact with the LC is addressed. Finally, a possible way in which the student would be able to teach the companion is discussed.

*Sponsored by Consejo Nacional de Ciencia y Tecnología (CONACYT), México.

Expertise and Behaviour

Work in LCSs has increased in the last few years but much remains to be done to explore the full capabilities and possibilities of these systems. Self and his colleagues proposed ITSs which offered collaboration with the student rather than instruction (Gilmore & Self, 1988; Cumming & Self, 1989). But it was Integration-Kid (Chan, 1991) the first system built as an LCS that introduced the idea of a LC. The LC in this system was capable of collaborating and competing with the student. Integration-Kid proved the feasibility of LCSs and demonstrated that they stimulated learning interactions which are not possible with a tutor only. But, more importantly, it raised the issues of the expertise and behaviour that such a companion agent should have if it is going to be of value to the student.

In Integration-Kid the companion had a knowledge level which is average to students in its domain. With this expertise the companion was thought to display sub-optimal performance which was expected to motivate the student by showing her that this is normal when learning. Integration-Kid behaved as a collaborator or as a competitor. These two behaviours were intended to help the student when solving a problem and to make her reflect when presented with different solutions. Hietala and Niemirepo (1997) have explored the effects of different degrees of companion's expertise. Their work used companions with low expertise (weak) and with high expertise (strong). Their interest was to select the companion's expertise to maintain the student's motivation to collaborate with the LC. Their experiments took into consideration the students' general learning capabilities and their personality traits to observe the effect and acceptance of the LC. In general, they found that students preferred strong companions specially when tasks got harder. The LCs in this work were there to give advice to the student and to collaborate with her. Both of the above LCSs try to help the student in her learning activities. They encourage collaboration and suggestions from the LC to help the student reflect on her knowledge. Competition gives the student the opportunity to see a different approach to do the same task and may stimulate her to work harder.

Another helpful way for a student to learn when interacting with a peer is *to teach her peer*. Research has shown that students who teach other students learn more and better (Berliner, 1989; Goodlad & Hirst, 1989; Palthepu, Greer, & McCalla, 1991). A student who needs to teach other people will have to revise, clarify, organize, and reflect on her own knowledge in order to be able to teach, i.e. the student will need to master the knowledge. A learning companion with less knowledge than the student should in principle be helpful for the student to learn by teaching. Therefore, we suggest that *a weak LC would help the student to learn better by encouraging her to teach it*. Although Hietala and Niemirepo found that strong companions were more used, the companions in their environment had 'fixed' knowledge — 2 strong and 2 weak companions. The student was able to select one of those 4 companions but she was not able to modify the companion's knowledge. Therefore, it was natural that when subjects were faced with more difficult tasks they preferred, in general, to get help from a more knowledgeable companion.

Implementation

An LCS is being developed to explore the idea of teaching a weak LC. The domain of the system is Binary Boolean Algebra. The tutor will teach the laws and theorems (rules) of this domain, how to use them, and when. Emphasis will be on teaching when each rule is best used to simplify a boolean expression. Even though it is a complex problem to decide which rule is best to use at any given moment, the system will give general guidelines which could be used successfully with many problems. The goal is to introduce students to Boolean Algebra and to the basis of boolean simplification.

As mentioned before, the tutor will be the one in charge of teaching both learners — the companion and the human student. Because the main objective of the system is to explore the interaction between the companion and the student, the tutor is being designed to be as unintrusive as possible. Its tasks are to teach concepts to students, to give examples using those concepts, to select problems for the students, and to comment on the students' performance. Only when the tutor is doing one of these four activities will it be in contact with the students. Once it has given a problem to them, it will 'disappear' from the students sight and its task will be to monitor them while solving the problem. In this way, the interaction could focus on that between the student and the companion. After they have finished solving the problem, the tutor will comment on their performance. This cycle continues until the end of the curriculum.

The learning companion is the most important part of the system. It will be implemented using simulation techniques. This means that it will not actually learn while the tutor is teaching. The companion's knowledge will be secretly selected by the tutor depending on the problem and necessities of the moment. Actually, the tutor will be controlling the companion throughout the complete interaction. It could be seen as if the companion and the tutor were communicating with each other without letting the student know. For example, when both learners are working on a problem the tutor could tell the companion to ask a question to the student or to give a suggestion which is expected to help in the resolution of the problem.

The student will be in control of the interaction between her and the companion. The tutor will present the problem making it clear that they have to work on it as a team. There is flexibility for the division of labour within the team. It is up to the student to decide who will actually solve it and if she will interact with the companion or not. The student will be able to choose not to interact with the companion if she wants. This freedom of interaction generates the problem of, how to motivate the student to interact with the LC — this issue is discussed below. If the student decides to interact with the companion, she will have the opportunity to teach it. But, how will the student teach the LC? A suggestion about how to do this is presented below.

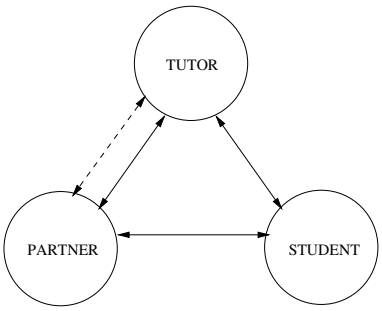

Figure 2: Tutor controlling LC.

Figure 2 presents the general agent architecture of the system being implemented. The solid arrows between the agents represent a communication between two of the agents that is seen by the third agent. The dashed arrow represents a private communication — in this case between the tutor and the companion.

Motivating the student

Interaction between the student and the companion is essential in any LCS. If at any time the student decides that she does not want to interact with the companion anymore, then either 1) the LCS becomes an ITS — if the student is able to continue working without interacting with the companion, or 2) the interaction with the LCS is terminated — if the student must use the LC. Both cases are detrimental to the objective of LCS so care should be taken with the motivation of the student while using the system.

In the implementation described before, the student is not told to interact with the companion. She will have the possibility to decide if she wants to make use of the companion or not. Even in the case where she decides to interact with the companion, there is a risk that after a while the student could get bored with a weak companion. A weak companion will give wrong suggestions and solutions some of the time. It will also appear to be very keen on being taught by the student by asking questions which show that it does not know very much. These actions could make the student see the LC as an annoyance, more work to do. She might get bored with it and decide not to use it anymore. Of course, the student may realize the possibilities for reflective learning provided by the companion and may not need further motivation to work with it.

Our system will have the flexibility of working in two interaction modes: motivated and free. These modes will help to observe if the student gets bored with a weak LC and if pressure on the student to interact with the companion is useful. In free interaction the student will only be told that using the LC is beneficial for her to learn — specially if she teaches the student. In motivated interaction the score mechanism in figure 3 will be used.

The aim of the scores is to motivate the student by giving her a challenge. The student will be presented with the three scores in figure 3. The objective is that the student finds a way to obtain enough points to advance a

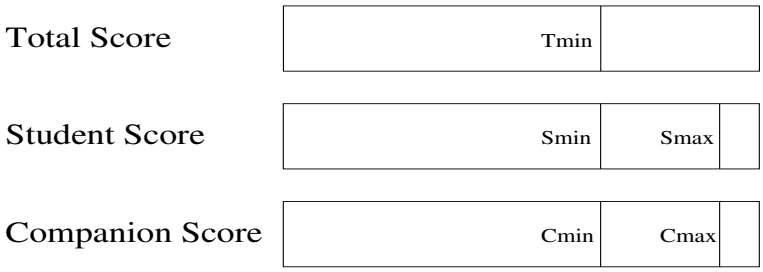

Figure 3: Motivation mechanism.

level in the curriculum. This will depend on the total score which is a combined score for both the student and the companion. Tmin marks the minimum number of points to advance a level. The other two scores are the scores for the Student and the Companion respectively. The minimum mark in these scores indicates the minimum number of points each student needs to be able to advance a level. The maximum mark is the maximum points that will be given at a particular level. By working on problems the solver(s) will get some points, whether solutions are correct or not. The key factor is that the companion's points will have more impact in the total score. This way of allocating points is intended to prevent the student advancing a level without having used the companion enough. It is expected that once the student realizes that the points obtained by the companion are better for the total score, the student will use more and more the LC. Once the student is interacting with the companion she will have the possibility to teach it. The scores should provide enough motivation for the student to make her want to interact with the companion in order to advance a level.

Teaching the LC

In the present LCS implementation the main activity between the student and the companion will be teaching. The weak companion will behave in a way which will encourage the student to teach him. For example, the companion could ask the student if she agrees with him or not in using a simplification rule. Very often during the interaction the companion's suggestions will be wrong, so as a way to disagree with him the student could decide to teach him. The companion could also ask the student directly to teach him. The option of teaching the companion will always be available when the LC is working on a problem.

To teach the LC the student will use a window such as the one in figure 4. This window is based on the idea of inspectable student models (Bull & Broady, 1997) in that it represents the knowledge of the companion at a particular moment. The window provides the student with a series of buttons and menus which let her communicate with the companion. It would be better if the student could discuss directly with the companion using some form of natural language, but unfortunately the present state of the art in this area is not enough to support the kind of dialogues which would be needed when teaching.

The teaching window presents the knowledge of the companion at a specific moment during the interaction. The objective is to let the student see exactly what the companion knows when trying to solve a problem. In figure 4, the companion's knowledge is represented by a list of simplification rules labeled 'Rule Order'. This list contains, in order of priority, all the rules that the companion knows how to use. In the figure, the companion tries to apply rule 'r1' first. If he can not apply this rule, he will try to use the next rule, 'r2'. He continues trying rules until one can be used.

When the student is teaching the companion, she will need to modify the list of rules in the way she thinks it is better to solve the current problem. In order for her to select the companion's new rule order she will need to understand why the LC is using that particular order. To understand this she will first need to think about her own knowledge of the domain, i.e. what order does she use to simplify expressions and why. In summary, the student will need to revise, clarify, organize and reflect on her knowledge before she can teach the LC — by changing the order of the rules.

The menus and buttons in the window will let the student modify the rule order in which the companion is using the rules. Once she has taught the companion, it will use the rules as it has been told. The student can

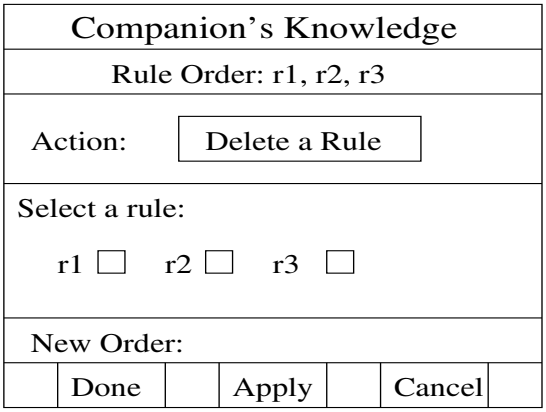

Figure 4: Teaching window.

then appreciate if what she taught the companion is appropriate or not. The student can continue teaching the companion as much as she likes. In a sense, by teaching the companion, the student will have the possibility to experiment in the domain and learn more about it.

The issue of exactly which rules the companion should be primed with secretly by the tutor remains to be solved. One idea is to make the companion's knowledge track that of the student, but some steps behind. In this case the student would effectively be reflecting on her own past performance.

Conclusion

LCS are systems which try to resemble human social environments. The learning companion adds a new dimension to the interaction between computer and users. It is claimed that the benefits of having such a companion are similar to the benefits of having a human peer. However, research must be done in order to understand the actual benefits of a computer companion. In particular, the effect on the student of the companion's knowledge level and behaviour should be studied.

This paper describes a system being implemented that proposes to use a learning companion which has less expertise than the student. The aim is that the student would be able to reflect on her own knowledge by teaching the learning companion. A proposal to motivate the student to interact with the LC is offered. It is essentially challenging the student by giving her a goal in the form of scores. To teach the learning companion a teaching window is proposed which is based on the concept of inspectable student models. This window should help the student to reflect on her own knowledge.

References

Berliner, D. (1989). Being the teacher helps students learn. *Instructor*, *98*(9), 12–13.

Bull, S., & Broady, E. (1997). Spontaneous peer tutoring from sharing student models. In du Boulay, B., & Mizoguchi, R. (Eds.), *Artificial Intelligence in Education: Knowledge and Media in Learning Systems*, Vol. 39 of *Frontiers in Artificial Intelligence and Applications*, pp. 143–150. International Society for Artificial Intelligence in Education, IOS Press. Proceedings of AI-ED 97 World Conference on Artificial Intelligence in Education.

Chan, T.-W. (1991). Integration kid: A learning companion system. In Mylopolous, J., & Reiter, R. (Eds.), *IJCAI-91, Proceedings of the Twelfth International Conference on Artificial Intelligence*, Vol. 2, pp. 1094–1099. Morgan Kaufmann Publishers, Inc.

Chan, T.-W., & Baskin, A. B. (1988). "studying with the prince" the computer as a learning companion. In *ITS-88: Inteligent Tutoring Systems*, pp. 194–200 Montreal, Canada.

Chan, T.-W., & Baskin, A. B. (1990). Learning companion systems. In Frasson, C., & Gauthier, G. (Eds.), *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, chap. 1, pp. 6–33. Ablex Publishing Corporation, Norwood, New Jersey.

Cumming, G., & Self, J. (1989). Collaborative intelligent educational systems. In Bierman, D., Breuker, J., & Sandberg, J. (Eds.), *Artificial Intelligence and Education*, Vol. 3 of *Frontiers in Artificial Intelligence and Applications*, pp. 73–80 Amsterdam, Netherlands. IOS. Proceedings of the 4th International Conference on AI and Education.

Gilmore, D., & Self, J. (1988). The aplication of machine learning to intelligent tutoring systems. In Self, J. (Ed.), *Artificial Intelligence and Human Learning, Intelligent Computer-Aided Instruction*, chap. 11, pp. 179–196. Chapman and Hall Computing.

Goodlad, S., & Hirst, B. (1989). *Peer Tutoring: A Guide to Learning by Teaching*. Kogan Page, London.

Hietala, P., & Niemirepo, T. (1996). Studying learner-computer interaction in agent-based social learning environments. In Brna, P., Paiva, A., & Self, J. (Eds.), *European Conference on Artificial Intelligence in Education*, pp. 386–392 Lisbon, Portugal. Colibri. Proceedings of Euro AI-ED96.

Hietala, P., & Niemirepo, T. (1997). Collaboration with software agents: what if the learning companion agent makes errors?. In du Boulay, B., & Mizoguchi, R. (Eds.), *Artificial Intelligence in Education: Knowledge and Media in Learning Systems*, Vol. 39 of *Frontiers in Artificial Intelligence and Applications*, pp. 159–166. International Society for Artificial Intelligence in Education, IOS Press. Proceedings of AI-ED 97 World Conference on Artificial Intelligence in Education.

Palthepu, S., Greer, J., & McCalla, G. (1991). Learning by teaching. In Birnbaum, L. (Ed.), *The International Conference on the Learning Systems*, pp. 357–363. Association for the Advancement of Computing in Educatio (AACE). Proceedings of the 1991 conference.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

Debugging as program comprehension

Pablo Romero Mares
juanr@cogs.susx.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

1 Abstract

This paper reports an exploratory empirical study on Prolog debugging. This study focused on developing a descriptive characterisation of the debugging strategy that six subjects used in order to debug a small Prolog program and also on identifying possible research themes in the area of teaching program debugging. The study identified several debugging techniques that subjects commonly apply when doing program debugging. This debugging behaviour is similar to the debugging strategies that the literature on this area reports for the case of other languages. Two of these techniques, gathering information and visual inspection of the code, seem to be very important for the debugging task. These techniques are described in detail and their relation to a specific debugging framework is discussed. Finally, some research themes emerging from these debugging techniques are also discussed.

Keywords: empirical studies of programming, program debugging, debugging strategy.

2 Introduction

Debugging is a central skill in programming and yet is rarely taught explicitly. This seems to be mainly due to the fact that our knowledge about debugging as a cognitive process is still rather vague.

The study described in this paper set out to see whether the models and techniques described in the literature for procedural languages also applied to Prolog. In particular it was concerned with whether the debugging techniques reported in the literature for the case of procedural languages were similar to those applied for Prolog and also with whether there was any resemblance between a specific debugging framework proposed by Brna, Bundy, and Pain (1992) for the case of Prolog and the debugging behaviour of Prolog programmers.

This document is divided into three sections. The next section gives a brief account of studies and experiments about debugging. The fourth section describes the experiment settings. The final section talks about the experiment results, discusses its findings and mentions possible research themes.

3 Studies of debugging

Studies of program debugging have concentrated mainly on procedural and functional languages and have developed several descriptive characterisations of the debugging task. This section describes these models and their associated studies. It also reviews a proposed framework for the specific case of Prolog.

Studies of debugging have concentrated mainly in three aspects of debugging: developing an overall model of the debugging task, obtaining a characterisation of the different activities that programmers perform when debugging and detecting the differences in behaviour and performance among novices and expert programmers.

The debugging task has usually been considered as an specific case of fault finding. As an instance of this activity, overall models proposed for debugging share many characteristics with those proposed for other fault finding activities like electronics troubleshooting or medical diagnosis. These models propose that the debugging process consists mainly of four subtasks: code comprehension, bug detection, bug localization and bug repair (Kessler & Anderson, 1986; Katz & Anderson, 1988). Code comprehension has been identified with the task of understanding

what the program does. It has been assumed that this task is performed through code reading episodes. The bug detection subtask consists in somehow testing the program to find out any possible discrepancies in the correct and the actual output. The bug localization subtask consisted in actually finding the piece of code responsible for the error. The last subtask, bug repair, has been identified with correcting the code. Although there are specific differences in the behaviour of subjects, this model of debugging has been said to capture the main steps of the debugging process that programmers perform.

Another point of view is that debugging should be considered as a component of software design rather than a fault finding activity (Gilmore, 1991). In this view, comprehension does not necessarily precede bug location. Instead, debugging occurs as part of an on-going comprehension process. Also, this view considers that there is not a single debugging process, but that programmers can choose to perform the debugging task in several different ways.

There are other studies that, while not proposing an overall model for debugging, describe the techniques and strategies programmers applied to debug programs. A list of the most commonly reported strategies includes: 'visual inspection of the code', 'hand simulation', 'gathering information', 'topographic search', 'symptomatic search' and 'trial and error'. The first of them, visual inspection of the code (Jeffries, 1982), seems to stand for trying to build a mental model of the program structure through code reading. The next activity, hand simulation (Katz & Anderson, 1988; Gugerty & Olson, 1986; Jeffries, 1982; Eisenstadt, 1993), means that the subject tries to execute the code as the machine would. Gathering information (Jeffries, 1982; Eisenstadt, 1993) is reported as a collection of techniques which aim at collecting information from the program behaviour about the error. Topographic search (Katz & Anderson, 1988; Gugerty & Olson, 1986), as mentioned before, has to do with using clues in the output or testing the internal program state to narrow the possible location of the bug to a small region of the program. In symptomatic search (Gugerty & Olson, 1986; Eisenstadt, 1993), the programmer makes use of prior debugging knowledge and recalls a bug that has previously caused symptoms similar to the current ones. The last strategy mentioned, trial and error, means making changes to the code without a clear idea neither of the problem nor the consequences of these changes.

It was mentioned that another point of interest in debugging has been the differences in behaviour and performance between novices and experts. The main findings in this area have established that expert perform better than novices and that experts have a superior ability to comprehend programs and to see the code as a hierarchical structure of meaningful segments (Jeffries, 1982; Vessey, 1985; Gugerty & Olson, 1986; Nanja & Cook, 1987).

It has to be noted that the majority of these studies aimed mainly at developing a descriptive account of the activities that programmers seem to follow to as part of the debugging task. Also, as was mentioned before, they focus on languages like Lisp, Pascal, Cobol or Logo, but not on Prolog. One of the few studies about debugging in Prolog is by Brna et al. (1992). Brna et al. developed a framework for debugging based on a four level bug description. These four levels are: *symptom* description, *program misbehaviour* description, *program code error* description and *underlying misconception* description. *Symptom* description is the description that the programmer gives in terms of the behaviour of the program. *Program misbehaviour* description is also a symptomatic description in terms of the program behaviour, but taking into account the different modules that comprise the program. The *symptom* description normally describes the behaviour of the program in terms of the answer given by a call to the main predicate. The *program misbehaviour* description, on the other hand, is related to the behaviour of the answer given by a call to the main predicate, but also to the answer given by its problematic subgoal(s). This inspection of a subgoal's behaviour in the execution tree finishes until terminal nodes (subgoals) are reached. *Program code error* description is the explanation offered in terms of the code itself. Such a description normally suggests the correction to the code. *Underlying misconception* description is the possible fundamental misunderstandings or false beliefs that the programmer has about programming concepts. According to this framework, debugging can be seen as the process by which a programmer goes from the *symptom* description to the *program code error* description via the *program misbehaviour* description and then, if necessary, on to the *underlying misconception* description. Brna et al. give a detailed description of a debugging strategy that a programmer could follow to perform this process for the case of non-terminating programs (programs that take so long over processing that they appear apparently stuck in some computation).

Although Brna et al. claim that their proposed framework captures the main tasks of the debugging process, they only give a detailed description of the strategies programmers could use to go from the the *symptom* description to the *program misbehaviour* description. The general strategy that this framework assumes programmers follow to do this is shown in figure 1.

- Turn on the trace.
- Issue the goal.
- *creep* to examine the subgoals.
- *skip* over each subgoal.
- If an incorrect result is detected, *retry* the last subgoal.
- *creep* to examine the behaviour of the defective subgoal's subgoals.
- Repeat the process for the new set of goals.

Figure 1: General strategy to go from the *symptom* to the *program misbehaviour* description
From Brna et al. (1992)

It can be noted that this strategy is based on a top-down methodology for the debugging task. This top-down methodology assumes that programmers will try to find errors by querying the main predicate of the program and will advance querying subgoals until they find the problematic procedure. This strategy assumes that programmers will mainly apply a data gathering technique when debugging a program.

Brna et al. give a detailed description of the strategy to debug non-terminating programs. For the purpose of comparing the debugging behaviour of subjects in the experiment reported in this paper with the debugging strategy for non-terminating programs, the end of this section talks about the sub-case of Malignant Endless Building.

Brna et al. subdivide the non-terminating program problem into different categories. One of these has to do with the case in which, through a step trace of the program, subgoals for the query under current investigation (which is apparently non-terminating) keep appearing. This subgoal calling might be an infinite process. This category of non-termination is known as *Endless Building*, and if this process is indeed infinite the sub-case is called *Malignant Endless Building*. There are two proposed ways to detect a case of *Endless Building* and one specific strategy for the case of *Malignant Endless Building*.

The first way to detect a case of *Endless Building* is to examine the ancestors in the step tracer information looking for a repeated goal. Finding a repeated goal would be evidence of this kind of problem. A further test could be to examine the recursion argument of the problematic call to see whether it is 'decreasing' healthily. This means watching the argument whose tail is processed in the recursive call. This argument is expected to 'decrease' through every recursive call. If this is not the case, then, a case of *Endless Building* is very likely to be happening.

One specific strategy to test a case of *Malignant Endless Building* would be to use the standard tracer to skip, retry and interrupt by hand calls to the suspicious procedure.

4 Experiment

4.1 Aims

The aims of the experiment reported here were to obtain a descriptive characterisation of the debugging techniques applied by Prolog programmers and also to find out whether there were any differences between the debugging behaviour of subjects that performed the debugging task in a successful way and those who were not so successful.

4.2 Subjects

Six Computer Science PhD students were the subjects of the experiment. These subjects had taken at least one introductory course on Prolog. Two of them were using Prolog in their project and one of them was an assistant for introductory Prolog courses. All of them had experience with other programming languages, mainly procedural ones.

```
component(earpiece, e32).
component(mouthpiece, p24).
component(cord, sc27).
component(body, bb49).

unit(handset, [earpiece, mouthpiece]).
unit(phone, [handset, body, cord]).

list_components(Component, List) :-
        component(Component, List).

list_components(Component, FList) :-
        unit(Component, CList),
        process(CList, Flist).

process([],X).

process([Component|Tail], Full_list) :-
        list_components(Component, Comp1),
        prcess(Tail, Rest),
        append(Comp1, Rest, Full_list).

append([X|Xs], Ys, [X|Zs]) :-
        append(Xs, Ys, Zs).

append([],Zs,Zs).
```

Figure 2: Buggy code for the Phone Components program

(From Mulholland (1995))

4.3 Materials

The program to be debugged was a small 19 line Prolog program that performed a simple database retrieval of telephone part codes. This example program was used by Mulholland (1995) to compare different debugging tools in terms of their effectiveness and also in terms of the debugging and program comprehension strategies that they encouraged. Mulholland claims that the four bugs contained in this program are representative of the kinds of errors that could be reasonably expected in such a program. The program with its four bugs is shown in figure 2.

The first bug is a data-flow error that consists in the first clause of the *list_components/2* predicate returning an atom and not a list. The effect of this error is not immediate and is that this atomic variable is passed as a first argument to *append/3*. This predicate requires all its arguments to be lists, so it fails. The second bug is a typo that produces a dataflow error with not so immediate effects. It is located in the second clause of the *list_components/2* predicate. The second argument of this clause has two names (*Flist* and *FList*). So, this clause returns a variable which is not instantiated, but this does not give any immediate problems until either the append predicate gets into

Errors / subjects	S1	S2	S3	S4	S5	S6
*list_components/*2a	f	c	c	c	c	
*list_components/*2b			c	c		c
*process/*2a			c	c	c	
*process/*2b	c	c	c	c	c	c

- (f) found
- (c) found and corrected

Table 10: Errors and subjects who found and corrected them

trouble because of this uninstantiated variable or the program returns an open list as wrong output (the problem produced depends on the test data). The third bug is a base case error that produces a wrong instantiation. It is located in the base case clause of *process/2*. The second argument never gets instantiated. This produces an open list as a result. The last error is a typo. It is located in the second clause of the *process/2* predicate, when the recursive call is made. This error makes the program execution stop because of an apparent missing predicate. For referencing purposes, the first error will be referred as the *list_components/*2a error, the second one as the *list_components/*2b error, the third as the *process/*2a error and the fourth as the *process/*2b error.

It should be noted that the effects these errors cause can be different in the actual debugging sessions depending of the code alterations made prior to focusing in a specific bug and to error combination effects.

4.4 Procedure

Subjects performed the debugging task on-line and individually. They were given a task description that included the problem description and some sample calls to the main predicate of the program as well as the desired answers to these queries. They worked in a Prolog interpreter environment but could also change and reconsult the program. Subjects were told neither the nature nor the number of the errors the program contained. They were informed that they could perform the debugging task at their own pace and modify the program as much as they judged necessary. The debugging session had a limit of one hour.

During the debugging session the following data were collected: the subjects' verbal protocols, the different versions of the program they reconsulted in the interpreter and a history of their interpreter interaction (the queries they submitted to the interpreter).

4.5 Results

The findings that comprise the results of this experiment include: a descriptive success rate of the subjects, the different debugging techniques that they applied and an account of the behavioural differences observed between successful and unsuccessful subjects.

From the six subjects who participated in the experiment, two found and corrected the four errors the program had, one corrected three errors, two corrected only two errors and one corrected only one error and spotted another but was not able to mend it. Table 10 shows which errors were discovered and corrected by which subjects.

4.5.1 Observed debugging techniques

The main debugging techniques observed in the experiment were hand simulation, visual inspection of code, data gathering, checking coding conventions and trial and error. From these, data gathering and visual inspection of code were by far the most frequently used by the subjects. Checking coding conventions means inspecting the code looking for specific mistakes in what is considered as standard coding conventions. For example, checking the code to see whether there were any typos, or also checking to see whether there were any segments of code which did not conform to standard coding conventions (for

example, having a variable that was never instantiated, or a base case for processing a list that did not consider the empty list as one of its arguments).

Data gathering was, as mentioned above, one of the most frequently used debugging techniques. This technique was performed through the application of queries to the Prolog interpreter. Subjects used queries to narrow the bug's possible location. Asking queries to the interpreter was an easy way to find out if a procedure was working correctly, or if it had some apparent problem. Therefore, programmers could ignore correct procedures and concentrate on those that had an undesired behaviour. Asking queries to the interpreter was also used to test hypothesis about the program behavior or to test the correctness of a modification to the program. Figure 3 shows part of the interpreter history annotated with the subject's comments before and after each one of the queries for subject S3. It can be observed that the first query is just of an exploratory nature. The second tests the first modification to the program. The third and fourth query are also exploratory. They are used to rule out of the search an apparently correct procedure. The fifth query gives a clue about the possible location of the bug in turn. Unlike the previous two, it fails, so the subject concentrates his search for the error in that procedure. At this stage the subject combined a *visual inspection of code* with a *checking coding conventions* techniques. He found that the *process/2* base case predicate did not consider the empty list as the output argument and decided to modify the program. The sixth and seventh queries test this modification.

Data gathering could also be performed with the help of the interpreter step tracer. In this case, subjects normally tried to understand control-flow or data-flow details of the program through the application of this technique. In doing so, programmers would concentrate on both the code and the output of the step tracer, trying to establish a relationship between them. Figure 4 shows part of the interpreter history annotated with comments before and after each one of the tracer interactions for subject S2. In this case, both tracer interactions are used to understand control-flow and data-flow issues.

As it can be noted in both examples, data gathering was normally combined with visual inspection of code episodes. Data gathering would influence the place in which subjects focus their visual inspection of code but also the queries they asked could be motivated by a program reading episode. Modifications to the program would normally occur after a visual inspection episode, but they could also be motivated by a combination of these two techniques, especially when data gathering was done with the help of the step tracer.

Queries could be written to test the main procedure or a specific subgoal. It seemed that when subjects made up a query to obtain information about a process, they knew which specific clauses would be executed by that query. So, if they wanted to try a specific set of clauses, they would make up a query to do it.

As mentioned above, visual inspection of code episodes were frequently applied by subjects. This debugging activity sometimes motivated changes to the code and the selection of the next query to apply. Subjects would mainly use this activity to gain an understanding of different aspects of the program code. However, some other activities, sometimes in combination with visual inspection episodes, could be used to obtain this understanding as well.

Although visual inspection seems to be an important debugging activity, little can be said about it from this experiment. This was mainly due to the fact that no provision was made to record this activity in a more fine-grained way. It would be interesting to know whether there are specific patterns that successful programmers follow to scan the code, if there are sections of the code that they watch first, in which parts of the program they focus their attention more frequently and which kind of information they extract from which parts of the code.

```
‘well I’ll first test one of the queries of the handout, the
first. Just to see what happens.’

?- process([earpiece,mouthpiece],L).
no

‘Process says no. Ok, so I see process looks like is a recursive
procedure. Oops, I see a typo in process second clause, so instead of
process you’ve got prcess, so let’s change that, load that again, and
test it again.’

?- process([earpiece,mouthpiece],L).
no

‘no, so also it seems that append is defined with the base case at the
bottom. Append again is recursive, and it looks like’

?- append([a,b,c],[d,e,f],L).
L = [a, b, c, d, e, f] ?
yes

‘right append looks ok. So may be list_components, ok, so
list_components tries to find out whether the first component is just
a component or whether it can be broken down, for example handset
which is earpiece and mouth piece. Test list_components, first of all
just with earpiece,’

?- list_components(earpiece, L).
L = e32 ?
yes

‘and now with the handset.’

?- list_components(handset,L).
no

‘Aha!, so list_components fails with a handset, so its the second
clause with handset. That does a recursive call to process... This
stopping condition is slightly worrying for process,  because given
an empty list I guess it should return an empty list, but is
undefined. So, lets change this condition... and I’ll try
list_components again.’

?- list_components(handset,L).
no

‘No, right. Ok, so lets test the stopping condition...’

?- process([],L).
L = [] ?
yes
```

Figure 3: Annotated interpreter history for subject S3

```
‘...I don’t understand the process procedure base case. I’ll try with
an spy in list_components and a simple query’

?- process([earpiece],L).
** (1) Call : process([earpiece], _1)?
** (2) Call : list_components(earpiece, _2)?
** (2) Exit : list_components(earpiece, e32)
** (3) Call : process([], _3)?
** (3) Exit : process([], _3)?
** (3) Redo : process([], _3)?
** (3) Fail : process([], _3)?
** (2) Redo : list_components(earpiece, e32)?
** (2) Fail : list_components(earpiece, _2)?
** (1) Fail : process([earpiece], _1)?
no

‘list_components, and should enter here. Goes out of the base case
and it looks ok. An then, it fails, here, when it returns from
list_components, and then it makes the recursive call. In this simple
case, tail is the empty list, but then when the recursive call is
made, it fails. When it returns it should call the append, but why
not? ... I’m going to make an spy of append, I didn’t want to. And
I’ll try this simple call again.’

?- spy append.
Spypoint placed on append/3
yes

?- process([earpiece],L).
** (1) Call : process([earpiece], _1)?
** (2) Call : list_components(earpiece, _2)?
** (2) Exit : list_components(earpiece, e32)?
** (3) Call : process([], _3)?
** (3) Exit : process([], _3)?
** (4) Call : append(e32, _3, _1)?
** (4) Fail : append(e32, _3, _1)?
** (3) Redo : process([], _3)?
** (3) Fail : process([], _3)?
** (2) Redo : list_components(earpiece, e32)?
** (2) Fail : list_components(earpiece, _2)?
** (1) Fail : process([earpiece], _1)?
no

‘Yes, it calls append, but after that it doesn’t do anything. Append,
uhm, oh, lets see, append fails because I can’t do an append of, well,
append could fail because of two things. Because append is supposed to
work with two lists and I don’t have two lists...’
```

Figure 4: Annotated tracer history for subject S2

```
'So the first thing I'll do is just see what works. Let's try
something else. Ok, it's gonna give no for everything so I'll look at
the program. Process, two clauses, process list, the empty list will
get at, uhm. So let's try that [process/2]. Process, empty list,
L. Oh, so process component, tail, so there is a mistake here
[process/2b error] so that's something. May be I'll check for more
mistakes... Something I half remember, I'm not sure if it is right or
not. I'm sure the stopping clause has to come first, fairly sure...'
```

Figure 5: Fragment of subject S6 comments

4.5.2 Differences between successful and unsuccessful subjects

One important point when considering the explicit teaching of debugging is to observe the differences in debugging behaviour between successful and unsuccessful subjects. Apparently, successful and unsuccessful subjects applied similar techniques. However, there were differences in aspects such as: which techniques were most common among each of these subgroups, the results of applying specific techniques, the quality of the structural knowledge they have about the language, the interpreter tools and the notional machine, how global or local the comprehension process was and which parts of the task they focused on.

There were some techniques that were applied more frequently among the successful subjects. These techniques are *visual inspection of code* and *checking coding conventions*. In general, successful subjects would tend to apply visual inspection of code episodes frequently, mostly at early stages of the debugging session. Subjects would use these episodes to form an early, rough idea of what the code was doing. Successful subjects were able to put into words this early mental model they formed. Most of the time their claims represented a good approximation to the code structure and behaviour. These subjects would also apply checking coding conventions episodes to spot errors before coming across their symptoms. It seems that their strong knowledge of the language syntax and coding conventions allowed them to apply this technique. Less successful subjects would also spot apparently suspicious segments of code, but they would base these findings on erroneous beliefs about the language and its coding conventions. The claims about the *list_components/2* procedure and the finding that the *process/2* base case do not seem to conform to standard coding conventions for the case of subject S3 (figure 3) can be compared to the comments of subject S6 (figure 5). It can be observed that this subject did not attempt to externalise his beliefs about what the program computed. Instead, he engages in testing the procedures through queries to the interpreter. Also, he expresses an apparently false belief about the base case predicates. He thinks that base predicates have to come first in procedure definitions.

The quality of knowledge about the interpreter and notional machine were another advantage that more successful subjects had over less successful ones. The later knew how to interpret the information from the step tracer correctly, while the former were sometimes misguided by a wrong interpretation of this information. Subject S2, for example, believed that the *redo* and *fail* messages indicated the same thing, the failing of a process call.

Another important difference has to do with how to modulate the level of understanding of the program code. This understanding could be global or specific to some particular procedure or clause. Successful subjects normally knew how to modulate this comprehension process, so that it would be both general and specific enough. It needed to be general to allow a basic understanding of the program behaviour and structure but also specific to search for suspicious parts of code. Perhaps one of the facts that allowed successful subjects to gain this dual understanding was the early mental model of the

```
'Ok, the first thing I have to do is to examine the program to see if
it meets the specifications that are written in the hand out. The
components are stated ok, the parts of telephone, and after that the
unit. Ok, they form a hierarchy. Telephone is made by handset, body and
cord and handset is made of earpiece and mouthpiece. No problem...'
```

Figure 6: Fragment of subject S5 comments

program that they were able to build. It seems that through this early understanding, among other factors, they were able to guide the search for specific segments of code.

Finally, another relevant difference seems to be that successful subjects would relate the program code and behaviour to the problem specifications. It can be observed in the comments in figure 6 that subject S5 tries to relate this two sides of the problem task. Unsuccessful subjects would rarely engage in these kind of processes, they normally were more concerned with trying queries or spotting the errors.

5 Discussion

The main findings of the experiment are that subjects performed very similar activities to the ones reported in the debugging literature, and that although the general strategy that they followed seems to have some common points with the debugging framework proposed by Brna et al., this framework seems to be too limiting for the range of techniques that programmers applied. Another important point is that there were differences in both the strategical and structural knowledge between successful and unsuccessful subjects.

Most of the debugging techniques that subjects applied are similar to the ones reported in the debugging literature for the case of procedural and functional languages. Subjects indeed applied hand simulation, visual inspection of code, data-gathering and trial and error. They did not use symptomatic search, but they applied a similar technique: checking coding conventions. The similarity between these two techniques is that both use advanced programming knowledge to recognise typical errors. The difference resides in that while the former uses the program behaviour to spot the error, the later finds it focusing on the program structure.

5.1 Comparison to Brna et al.'s framework

The previous section showed that queries were used to narrow down the bug's possible location. This application of queries bears some resemblance to the method Brna et al. propose as a solution for debugging Prolog programs. This section compares the debugging behaviour displayed by the subjects with Brna et al.'s proposed framework. Although it is not expected that the subjects of this experiment behaved exactly as this framework propose, the objective of this comparison is to establish whether this proposed framework has some cognitive plausibility. First, this section compares the general debugging strategy proposed in this framework with the one followed by the programmers in this experiment and then it analyses the case of non-terminating programs in more detail.

As mentioned above, Brna et al. propose a general strategy that programmers could use to go from the the *symptom* description to the *program misbehaviour* description. This strategy requires programmers to follow a top-down approach when using a Prolog tracer. This general strategy is shown in figure 1. The first difference between this strategy and the behaviour displayed by the subjects of this experiment is that these programmers did not exhibit a sophisticated use of the standard tracer. When using the step tracer, subjects would only use the default *creep* option. This might be due to the fact that

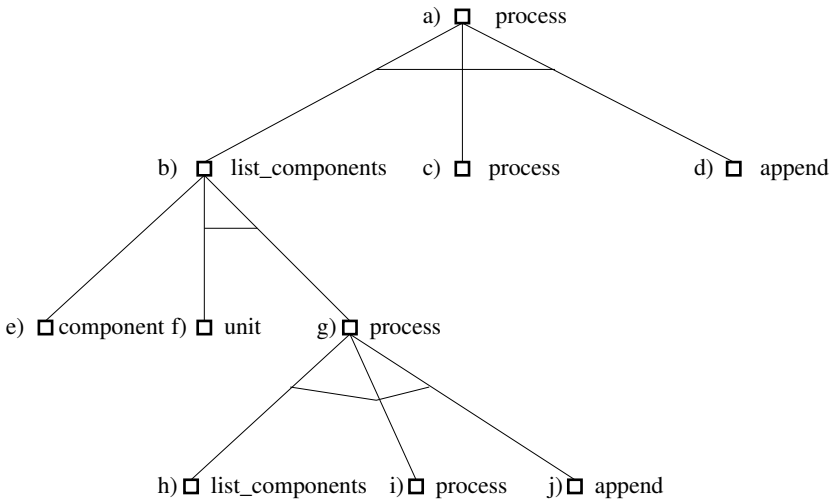

Figure 7: AND-OR tree for a query to *process/2*.

they were debugging a small program and the full tracer information was still manageable for them. Of course another option is that they did not know how to use other options of this step tracer. However, the queries they submitted to the interpreter bear some resemblance to the idea of top-down debugging proposed in this framework. It can be seen in the interpreter history in figure 3 that this subject tried a call to the main procedure. As it failed, he made up queries to test its subgoals. This behaviour is in agreement with this top-down approach. However, this was not always the case. It can be seen in this interpreter history that the decision to try the last two queries is because of the correction of a suspicious segment of code discovered by a visual inspection of the code. So, it seems that the difference between the strategy that this general framework proposes and the subjects' behaviour is that it cannot be assumed that data gathering is the only or the main activity that is performed during debugging. As it has been mentioned, subjects applied a variety of activities: visual inspection episodes, data gathering in the form of simple queries or queries and step tracer, inspections to the code looking for specific errors, hand simulation and combination of some of these. It is possible that selecting the next query to apply could have been motivated by any of these activities. So, although subjects seemed to be following a strategy similar to the one proposed by Brna et al.'s framework, this framework does not seem to consider that programmers have a variety of resources available, apart from data-gathering techniques, to perform the debugging task.

It has to be mentioned that the program used in this experiment was probably not the best example to investigate the possible patterns in the sequence of queries applied. The fact that it contained a double recursive call made it complex enough to observe a variety of debugging behaviours, but it made it difficult to relate the sequence of application of queries to the program calling structure. For example, it cannot be assumed that trying a query to the *process/2* clause would be made because the subject was trying to test the node *c)* in the AND-OR tree of the code in figure 7. The subject's idea might have been to test the node *i)* instead.

Another problem with this proposed framework is that, as Eisenstadt (1993) mentions, sometimes there is a considerable gap between the place in the code where the problem arises and where the problem is detected. Sometimes finding the predicate where the problem shows its symptoms does not say much about the place in the code where the problem was generated. Eisenstadt calls this phenomenon the cause/effect chasm. In terms of Brna et al.'s framework, the step from the *program misbehaviour*

description to the *Program code error* description might not be straightforward. In fact, if the gap between the place the error is generated and the place the symptom of the error appears, trying to obtain a complete *program misbehaviour* description might be misleading.

The debugging session of subjects S3 and S4 gave a chance to analyse the case of non-terminating programs. These two subjects corrected the *process/2*a, *process/2*b and *list_components/2*a errors and then tried a query to test the main procedure for the case of the handset. This resulted in a non-terminating call to the *append/3* procedure. This non-terminating call was due to the fact that *append/3* was called with an uninstantiated variable as first argument because the *list_components/2* second predicate was losing a variable (the *list_components/2*b misspelling error). The uninstantiated variable that *append/3* received as first argument is the one whose tail is processed in the recursive call. This means that this is the recursive argument that should 'decrease'. As an uninstantiated variable would always have a tail to send to the recursive call, the *append/3* procedure would implement a case of *Malignant Endless Building*. These two subjects applied neither the general strategy, the specific techniques to cope with the case of *Endless Building* nor *Malignant Endless Building*. Actually, they did not even realise that the problem symptom appeared in the call to the *append/3* procedure. When they tested the main procedure for the case of the handset and the problem arose, they went down in the AND-OR tree to test procedures several levels below the one they had queried. As they had no problems with these procedures, they went up the AND-OR tree, testing procedures between the ones they had just tried and the problematic one. In this way, and combining this data gathering technique with visual inspection of code episodes, they were able to find the problem without producing a non-terminating call again. Again, the main difference seems to be that subjects did not restrict their debugging behaviour to applying data gathering techniques.

As mentioned before, the main point in comparing the debugging behaviour of the experiment subjects with the proposed framework is not to verify if they matched exactly, but try to determine whether this framework is cognitively plausible. In trying to establish the cause of the observed differences two explanations seem worthwhile to explore: first, that programmers have more resources available to those that Brna et al.'s framework assume, and second, that programmers deviate from this framework because they cannot retain in their working memory all the infomation required to follow this approach. In any case, it seems that either the approach or its teaching implementation have to consider these differences. Teaching the debugging framework developed by Brna et al. could be sensible if it is taught as one of several possible resources that can be applied when debugging. An instruction that presents it as the only or main debugging strategy might not give the desired results.

6 Conclusions

This paper reported an empirical study of Prolog Debugging. The main findings of the experiment are that subjects applied a similar set of debugging techniques to the ones reported in studies for procedural languages, that the debugging framework proposed by Brna et al. seem to be too limiting for the variety of resources that programmers can apply when debugging and that there are interesting differences in the strategical knowledge between subjects who performed the debugging task successfully and those who did not.

The debugging framework proposed by Brna et al. seems to be too limiting mainly because it assumes that programmers rely a lot in data gathering techniques when debugging. They certainly seem to make use of it, but they also have several other techniques that can be applied at any time depending on the context. What seems certain is that in one debugging session programmers combine several techniques to the extent that it is difficult to say what is the impact of each individual technique on the result of the task.

The fact that subjects would normally combine several debugging activities at any moment of their

debugging session makes it difficult attempting to characterise each one of these activities separately. One solution to this problem could be to design experiments where subjects were forced to use only one or some of these activities (the ones that are going to be studied). However, it seems that the combined effect of these activities is more powerful than the simple sum of each one of them.

There were several interesting differences in the strategical knowledge applied between programmers who performed the debugging task in a successful way and those who were less successful. An important difference seems to be that more successful subjects had frequent episodes of visual inspection of code. These code reading episodes, together with their superior structural knowledge of the language allowed these subjects to have an early, global understanding of what the code was supposed to do. This early understanding also seemed to have allowed them to module their program comprehension process. They tried to construct a global comprehension of the code that would allow them to know in which parts of the program they should apply a focalised, detailed understanding.

It seems that code reading episodes are so important for program comprehension and also for program debugging, that they need to be studied in a closer way. Research in this area suggests that code reading is a search process, although the nature of the constructs searched for has not been established yet. It would be interesting to investigate whether visual inspection of code has any common points with program generation. It has been suggested that when generating code, programmers often build their solutions through recalling stereotypical segments of code, or plans, that are relevant to the case. Programmers generate code by recalling the focal parts of these plans and writing them first. Later, they fill in the other, less central parts of the code (Rist, 1989; Davies, 1994). It would be interesting to investigate if the comprehension process when applied through visual inspection of code episodes follow a similar procedure. This is, whether programmers try to make sense of the code also based on these plan constructs, and whether these central parts of plans are the starting point of this plan-based comprehension process.

A deeper understanding of the comprehension process that take place when debugging and how programmers develop expertise for this skill might offer interesting information about the way debugging can be taught and about important points to consider in the design of debugging tools.

References

Brna, P., Bundy, A., & Pain, H. (1992). A framework for the principled debugging of prolog programs: How to debug non-terminating programs. In Brough, D. (Ed.), *Logic Programming: New Frontiers*. Intellect, Ltd., London.

Davies, S. P. (1994). Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human Computer Studies*, *40*, 703–726.

Eisenstadt, M. (1993). Tales of debugging from the front lines. In *Empirical Studies of programmers, fifth workshop* Norwood,NJ. Ablex.

Gilmore, D. (1991). Models of debugging. *Acta psychologica*, *78*(1).

Gugerty, L., & Olson, G. (1986). Debugging by skilled and novice programmers. In *Proceedings of CHI'96, Human Factors in Computing Systems* Boston, Mass. USA.

Jeffries, R. (1982). A comparison of the debugging behaviour of expert and novice programmers. In *Proceedings of AERA annual meeting*.

Katz, I., & Anderson, J. R. (1988). Debugging: an analysis of bug location strategies. *Human-Computer Interaction*, *3*, 359–399.

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in lisp. In *Empirical Studies of programmers, first workshop* Norwood,NJ. Ablex.

Mulholland, P. (1995). *A Framework for Describing and Evaluating Software Visaualization Systems: a Case Study in Prolog*. Ph.D. thesis, Knowledge Media Institute, Open University, Milton Keynes, U.K.

Nanja, M., & Cook, C. (1987). An analysis of the on-line debugging process. In *Empirical Studies of programmers, second workshop* Norwood,NJ. Ablex.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*, 389–414.

Vessey, I. (1985). Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies*, *23*, 459–494.

In: Halloran, J. C. & Retkowsky, F. P. (Eds.). (1998). The 10th White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex. Cognitive Science Research Paper 478, School of Cognitive & Computing Sciences, University of Sussex.

KBANN's for Classification of Normal Breast ^{31}P MRS Based on Hormone–Dependent Changes During the Menstrual Cycle

Margarita Sordo, Hilary Buxton and Des Watson
{marias, hilaryb, desw}@cogs.susx.ac.uk

David Collins, Sabrina Ronen, Martin Leach and Geoff Payne
CRC Clinical Magnetic Resonance Research Group
email: {collins, sabrina, martin, g.payne}@icr.ac.uk

School of Cognitive & Computing Sciences
University of Sussex
Brighton
BN1 9QH

Abstract Knowledge-based neural networks (KBANN's) provide a means for combining knowledge of a domain in the form of simple rules with connectionist learning. This means that small sets of data (typical of medical diagnosis tasks) can be used to train the network as the initial structure is set and it is only necessary to refine these rules by training. This paper covers one such problem where Magnetic Resonance Spectroscopy (MRS) is used to investigate phosphorus-containing metabolites in the cell biochemistry of normal breast tissue. This study is part of an ongoing investigation into normal and abnormal breast physiology which may allow non-invasive detection of breast cancer. The hybrid methodology of the KBANN's approach allows the combination of empirical learning techniques with simple rules about dependencies. In particular here, the metabolites in the menstrual cycle are studied in the Knowledge Acquisition (KA) phase and the rules described. Then details of medical data and methodology are given together with results and conclusions. Finally, it is argued that KBANN's are a general and reliable aid in such tasks and can provide a sound base for further research into the metabolic changes found in breast cancer.

1 Introduction

The classification of 16 *in vivo* ^{31}P spectra into four possible phases of the menstrual cycle by means of knowledge–free and knowledge–based artificial neural networks (KBANN's) provides a real–life framework to asses the advantages of KBANN's over knowledge–free networks in a domain with such demanding constraints as scarcity and complexity of available data.

^{31}P Magnetic Resonance Spectroscopy (MRS), a non–invasive technique for the observation of phosphorus–containing metabolites and intracellular pH, plays an important role in the investigation of cell biochemistry. *In vivo* and *in vitro* ^{31}P MRS offer reliable means for detection of metabolic changes in the breast due to the influence of hormonal changes during the menstrual cycle. Previous observations with invasive methods indicate the necessity of a deeper knowledge of cyclic structural and metabolic changes of mammary gland that could guide to a sounder and clearer interpretation of breast physiology. At the same time, this information could allow a better understanding of subtle alterations in normal tissues, that could lead into the detection of breast cancer (Payne, Dowsett, & Leach, 1994; Twelves,

Porter, Lowry, Dobbs, & *et.al.*, 1994). To explore the full potential of such a promising non-invasive technique, it is necessary to develop a computational aid capable of classifying complex and limited data. KBANN's (Towell, Shavlik, & Noordewier, 1990; Noordewier, Towell, & Shavlik, 1991; Towell, 1991) are a reliable methodology that combines the strengths of both symbolic and connectionist approaches into a *hybrid* learning algorithm capable of learning from small data sets.

This paper presents results to assess the potential of knowledge–based over knowledge–free artificial neural networks for classification of *in vivo* ^{31}P MRS of normal breast tissues into four possible phases of the menstrual cycle. Classification is based on hormone–dependent changes in mammary tissue metabolism.

Knowledge required by the symbolic approach of KBANN's was extracted from the expertise of several authors in hormone–dependent metabolic and structural changes in normal breast tissues during the menstrual cycle. Relevant knowledge was elicited from a bibliographical review of important contributions to this field and was transformed into a knowledge base of hierarchically structured inference rules. This process was guided by a variation of an interviewing technique described by Kuipers *et al* (Kuipers & Kassier, 1987). Both knowledge base refinement (Fox, Myers, Greaves, & Pegram, 1987) and teachback interviewing (Johnson & Johnson, 1987) techniques were applied for further refinement of the knowledge base. Backpropagation learning algorithm (Rumelhart, McClelland, & *et al*, 1986b; Rumelhart, Hinton, & Williams, 1986a) was used for the empirical learning –connectionist– approach of KBANN's.

2 KBANN's

KBANN's are a *hybrid* methodology which combines explanation–based and empirical learning techniques. Strengths of one methodology overcome the weaknesses of the other, thus the combination of both approaches has a more robust performance than either method alone (Towell et al., 1990; Noordewier et al., 1991; Towell, 1991). This approach leans on the fact that living beings use previous knowledge to ease the learning of new tasks (Thrun, 1995).

KBANN's have two modules. The first part, the explanation–based module, consists in a set of approximately correct rules contained in the knowledge base. These rules will determine the initial topology of an artificial neural network: number of inputs, hidden and output units, connections between nodes, and associated weights and biases. The initial topology contains implicit knowledge of the problem domain, which prevents the network from learning from scratch. It also indicates the features believed to be important for a correct classification; reduces training time and, more importantly, diminishes the necessity of large training data sets. The empirical learning module consists of a neural network supervised learning algorithm which refines the knowledge embedded in the network's topology. A set of examples is presented to the network until it reaches an acceptable accuracy in its responses.

3 Normal Breast and Menstrual Cycle

Normal breast tissues in healthy pre–menopausal women are influenced by hormonal changes during the menstrual cycle. The secretion of hormones by the anterior pituitary gland and ovaries influence the mitotic activity of breast tissues. Histologic, structural and metabolic changes in the human female breast corresponding to different phases of the menstrual cycle have been observed, although the nature and extent of the response of mammary tissues to these variations in hormone concentrations are not yet clearly understood.

The menstrual cycle is completely dependent on the action of three gonadotropic hormones secreted by the anterior pituitary gland which are essential for the function of the ovaries. These hormones are

follicle–stimulating hormone (FSH), luteinizing hormone (LH) and luteotropic hormone (LTH). Based on endocrine events, the menstrual cycle can be divided into 3 phases: follicular, ovulatory and luteal phases. Figure 1 shows the changes in hormone levels during the menstrual cycle (Merck, 1997).

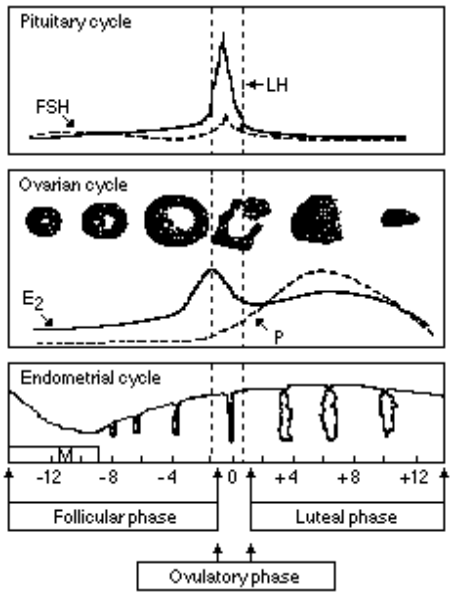

Figure 1: Hormonal changes in menstrual cycle

Follicular and luteal phases of the menstrual cycle are each 14 days long. During the follicular phase, the ovaries produce increasing amounts of oestrogen. In the first half of this phase, secretion of FSH slightly increases and causes the ovarian follicles to grow and develop. In the second half, secretion of oestrogen, particularly oestradiol (E_2), from the ovaries increases slowly and then accelerates and peaks during the day of LH surge. Just before the LH surge, progesterone levels also begin to increase. During the early luteal phase the *corpus luteum* supports the released ovum by secreting increasing quantities of progesterone and oestrogens, while LH and FSH levels drop down. During the late luteal phase, circulating levels of LH and FSH remain low, but begin to increase again with menstruation. Oestrogen and progesterone levels tend to decline in this part of the cycle. The ovulatory phase is 1–2 days long. During this time, a large amount of LH, called the preovulatory surge, is secreted by the pituitary gland and is necessary for the final follicular growth and ovulation.

Cyclical changes in the breast have been identified by invasive and non–invasive means. Both approaches have observed changes due to hormone influence. These changes are more likely to occur in the epithelium, rather than in stromal tissues, because epithelial cells are more sensitive to changes in oestrogen and progesterone levels (Payne et al., 1994; Longacre & Bartow, 1986; von Schoultz, Söderqvist, von Schoultz, & Skoog, 1996; Fanger & Ree, 1974; Potten, Watson, Williams, Tickle, Roberts, Harris, & Howell, 1988; Ferguson & Anderson, 1981; Twelves et al., 1994). Although these metabolic variations can be monitored with ^{31}P MRS, the complexity of the information prevents the expert from unveiling those subtle features embedded in a spectrum that can lead into the identification of certain patterns of cell behaviour.

It has been confirmed by several authors that in pre-menopausal women, maximal epithelial cell proliferation occurs during luteal phase of menstrual cycle (Strnad, Vachoušek, Čech, Smejkal, & Velek, 1995; Nazario, Simoes, & De Lima, 1994; Nazario, De Lima, Simoes, & Novo, 1995; von Schoultz

et al., 1996; Söderqvist, Isaksson, von Schoultz, & *et.al*, 1997; Drife, 1989; Olsson, Jernström, Alm, Kreipe, Ingvar, & *et al*, 1996; Potten et al., 1988; Ferguson & Anderson, 1981; Vogel, Georgiade, Fetter, Vogel, & McCarty, 1981)). *In vivo* ^{31}P MRS showed a relationship between PME levels and cell activity during the luteal phase of the menstrual cycle. In normal breast tissues, changes in PME levels are observed during periods of increased cell proliferation (Payne et al., 1994; Twelves et al., 1994). Thus, changes in mitotic and metabolic activity in normal mammary tissue during the menstrual cycle, can be monitored non–invasively by ^{31}P MRS.

The following section describes the method used to evaluate the performance of both knowledge–free and knowledge–based networks.

4 Knowledge Acquisition

Knowledge acquisition (KA) is a process that involves eliciting, analyzing, interpreting, formalizing and transforming into a suitable machine representation knowledge that experts use for solving problems (Kidd, 1987).

A set of statements with relevant information of metabolic and structural changes, due to variations in hormone levels during the menstrual cycle, was defined. The extracted statements where grouped into different categories: phase of the menstrual cycle a particular feature was related to, types of hormones that influence cell behaviour in normal breast, and changes in cell activity.

For each group, the statements were simplified. Similar statements were merged into a single statement. The simplified statements were transformed into a set of hierarchically structured *If...then* inference rules required by the knowledge–based neural network methodology (Towell, 1991). The final set of rules was the result of a process which involved the elicitation of knowledge from several authors, and the refinement of the knowledge base.

The knowledge acquisition process was guided by a variation of an interviewing technique (Kuipers & Kassier, 1987) since, in this case, the experts provided their knowledge through published work. A combination of knowledge base refinement (Fox et al., 1987) and teachback interviewing (Johnson & Johnson, 1987) was applied for the refinement of the knowledge base.

The final set of rules in the knowledge base used for classification of *in vivo* ^{31}P MRS of normal mammary tissues is presented as follows. A number was assigned to each rule. References attached at the end of each rule provide the sources knowledge was extracted from. Adjectives, such as *high* and *low*, referring to metabolite levels, cell proliferation rates or cell metabolic activity were eliminated. The decision of whether a metabolite level or cell activity was *high* or *low* was left entirely to the network, based solely on the value associated to that entry. Also, labels of this kind are more useful for binary inputs. They force the representation of knowledge into a more rigid and restricted format, which requires previous conceptualization from the user, reducing the generalization capabilities inherent to neural networks. Thus, for example, in the following rule: *If low* cell proliferation rate *then* early follicular phase, the network itself would decide if the value associated to the antecedent "proliferation rate" corresponds to a *low* level. If this is true, then the conclusion would be reached.

From the 17 rule knowledge base, rules 1, 2, 4 and 5 are the more widely accepted. Rules 6–17 are more specific. They are based on the data itself and the results presented by Payne *et al* (Payne et al., 1994).

R1: *If* (low) cell proliferation rate *then* early follicular phase. (Nazario et al., 1994; Olsson et al., 1996; Fanger & Ree, 1974)

R2: *If* (low) cell proliferation rate *then* late follicular phase. (Nazario et al., 1994; Olsson et al., 1996; Fanger & Ree, 1974)

R3: *If* (high) Pi *then* early follicular phase. (Pandya, Chandwani, Das, & Pandya, 1995)

R4: *If* (high) cell metabolic activity *then* early luteal phase. (Nazario et al., 1995; Longacre & Bartow, 1986; von Schoultz et al., 1996; Olsson et al., 1996; Potten et al., 1988; Ferguson & Anderson, 1981; Söderqvist et al., 1997; Vogel et al., 1981; Payne et al., 1994; Fanger & Ree, 1974)

R5: *If* (high) cell metabolic activity *then* late luteal phase. (Nazario et al., 1995; Longacre & Bartow, 1986; von Schoultz et al., 1996; Olsson et al., 1996; Potten et al., 1988; Ferguson & Anderson, 1981; Söderqvist et al., 1997; Vogel et al., 1981; Payne et al., 1994; Fanger & Ree, 1974)

R6 *If* (low) PME *then* early luteal phase. (Payne et al., 1994; Pandya et al., 1995)

R7: *If* (low) PME *then* late luteal phase. (Payne et al., 1994; Pandya et al., 1995)

R8: *If* (low) Pi *then* early luteal phase. (Pandya et al., 1995)

R9: *If* (low) Pi *then* late luteal phase. (Pandya et al., 1995)

R10: *If* (high) cell proliferation rate *then* late luteal phase. (Merck, 1997)

R11: *If* PME *then* changes in cell proliferation rate. (Payne et al., 1994; Twelves et al., 1994)

R12: *If* (low) PDE *then* late follicular phase. (Payne et al., 1994)

R13: *If* PDE *and* PME *then* biological changes. (Payne et al., 1994)

R14: *If* biological changes *then* early luteal phase. (Payne et al., 1994)

R15: *If* biological changes *then* early follicular phase. (Payne et al., 1994)

R16: *If* biological changes *then* late luteal phase. (Payne et al., 1994)

R17: *If* biological changes *and* PME *then* late follicular phase. (Payne et al., 1994)

These rules will later be translated into the appropriate notation. (See section 6).

5 The Data

16 *in vivo* ^{31}P MR spectra were obtained from four female premenopausal volunteers (ages 21–45), all with regular menstrual cycles and none using the contraceptive pill. Four ^{31}P spectra from each volunteer, one from each phase of the menstrual cycle, were acquired using a small surface coil positioned over the left breast, while the volunteer lay supine. Care was taken to avoid including signals from adjacent muscle. No loading correction was applied to the acquired spectra. Blood samples obtained the same day of each examination were analyzed for hormone levels (follicle–stimulating hormone (FSH), luteinizing hormone (LH), oestradiol and progesterone) to determine the phase of the menstrual cycle the spectrum was taken. After all the data were collected, each spectrum was assigned to the corresponding phase of the menstrual cycle: early follicular (EF), late follicular (LF), early luteal (EL) and late luteal (LL). Data were provided by the CRC Clinical Magnetic Resonance Research Group at the Royal Marsden Hospital, Sutton, and are the same cases as those used by Payne and co–workers (Payne et al., 1994).

6 Methodology

KBANN's require a knowledge base of hierarchically structured inference rules, features and consequents, not all necessarily to appear in the set of rules. The rules, written in a lisp-like notation, act as a starting point for the definition of the topology and connection weights of a feed-forward network. The set of rules can contain AND, OR and NOT relationships. The rules are assumed to be conjunctive, non-recursive and variable-free. (For a more detailed explanation, see (Towell, 1991; Sordo, 1997)). The knowledge base used for these experiments was as follows:

```
rule(1, ( (ef, ?x) <- (prolif_rate, ?x))).
rule(2, ( (lf, ?x) <- (prolif_rate, ?x))).
rule(3, ( (ef, ?x) <- (pi_level, value))).
rule(4, ( (el, ?x) <- (metab_act, ?x))).
rule(5, ( (ll, ?x) <- (metab_act, ?x))).
rule(6, ( (el, ?x) <- (pme_level, value))).
rule(7, ( (ll, ?x) <- (pme_level, value))).
rule(8, ( (el, ?x) <- (pi_level, value))).
rule(9, ( (ll, ?x) <- (pi_level, value))).
rule(10, ( (ll, ?x) <- (prolif_rate, ?x))).
rule(11, ( (prolif_rate, ?x) <- (pme_level, value))).
rule(12, ( (lf, ?x) <- (pde_level, value))).
rule(13, ( (bio_changes, ?x) <- (pde_level, value) &
                                (pme_level, value))).
rule(14, ( (el, ?x) <- (bio_changes, ?x))).
rule(15, ( (ef, ?x) <- (bio_changes, ?x))).
rule(16, ( (ll, ?x) <- (bio_changes, ?x))).
rule(17, ( (lf, ?x) <- (bio_changes, ?x) &
                       (pme_level, value))).
```

16 *in vivo* ^{31}P spectra from four normal breast from female pre–menopausal volunteers, four spectra from each volunteer, one for each phase of the menstrual cycle, were used for training the networks. In a preliminary stage, all the 16 cases were used as training set to evaluate the performance of knowledge-based networks with different topologies, as well as knowledge-free networks. *Leave-one-out* method, as it says, leaves one sample out of the training set. A network is trained with $n-1$ cases and then is tested with the remaining case. This process is repeated until the network has been trained with n possible sets. Its performance is the overall average of n trials.

7 Results

31 knowledge-free networks, with one to 20 hidden units were trained with different learning rate, momentum and initial random weight values. A very poor performance was observed in all of them, with an average pattern/error of 0.7500 and standard deviation of $1.2615x10^{-4}$. For the knowledge-based networks, 5 different topologies were created from 5 different sets of rules extracted from the main knowledge base mentioned above. All of them were trained with different learning rate and momentum values. Random noise was added for perturbing the initial weights and biases. Thus, there was a total of 42 networks for each topology. Both knowledge–free and knowledge–based networks had 7 inputs corresponding to the area under each peak of metabolites from *in vivo* ^{31}P spectra from normal breast tissues: PME, PDE, PCr, Pi, α–ATP, β–ATP and γ–ATP. They also had four outputs corresponding to each phase of the menstrual cycle: early follicular (EF), late follicular (LF), early luteal (EL) and late luteal (LL).

Average pattern/error and standard deviation, two measures to evaluate the performance of a network and set of networks with the same characteristics (e.g. topology, learning rate, momentum, initial random added noise), were used. The average pattern/error for a single network after training or testing is given by equation 8.

$$avg.\ pattern/error = \frac{total\ error}{number\ of\ patterns} \tag{8}$$

Similarly, the average pattern/error for a batch of networks is given by 9, where $avg.\ pattern/error_i$ is the average pattern/error for the network$_i$ given by eq. 8 and n is the total number of networks.

$$avg.\ pattern/error = \frac{\sum_{i=1}^{n} avg.\ pattern/error_i}{n} \tag{9}$$

The standard deviation (std) for a set of networks is given by equation 10, where $error_i$ is the average pattern/error of network$_i$, $avg.error$ is the total average pattern/error given by eq. 9, and n is the total number of networks in the batch.

$$std = \sqrt{\frac{\sum_{i=1}^{n} (error_i - avg.\ error)^2}{n-1}} \tag{10}$$

Table 11 shows how performance improves as the number of rules included in the knowledge base increases. A set of KBANN's with a topology defined from the whole set of 17 rules (section 4) had 0.1779 average pattern/error and 0.0269 standard deviation, whereas networks with knowledge bases of 11 rules (No. 3, 6–9, 12–17), 9 rules (No. 3, 8, 9, 12–17), 6 rules (No. 12–17) and 5 rules (No. 13–17) had average pattern/ errors of 0.2731, 0.2829, 0.3134 and 0.3662, and standard deviations of 0.0271, 0.0342, 0.0465 and 0.0286 respectively. These results, when compared to those from knowledge–free networks, clearly show how some added knowledge can positively influence a network's performance.

Trained Networks			
Type of Network	Features	Average pat/error	Standard deviation
Knowledge-free	No rules	0.7500	$1.2615x10^{-4}$
Knowledge-based	17 rules	0.1779	0.0269
Knowledge-based	11 rules	0.2731	0.0271
Knowledge-based	9 rules	0.2829	0.0342
Knowledge-based	6 rules	0.3134	0.0465
Knowledge-based	5 rules	0.3662	0.0286

Table 11: Knowledge–free and knowledge–based networks performance.

Based on their performance, the best three KBANN's, from the 17 rule knowledge base training batch, were selected. Two of them correctly classified the whole set of 16 cases with average pattern/errors of 0.0901 and 0.0207; the remaining network misclassified one case, with an average pattern/error of 0.0314. All of them have average pattern/error values well below the average 0.1779. These three networks, labeled KBANN-10, KBANN-12 and KBANN-13, were retrained and tested using the *leave–one–out* method (described in Section 6) to evaluate their performance over unknown cases. KBANN-10 was trained with learning rate= 0.5, momentum= 0.7 and seed= 1 (for the added random noise to initial weights and biases). It showed the best performance with an avg. pattern/error of 0.1335 (std= 0.1379) and 0.4380 (std= 0.3617) for training and testing respectively and correctly classified 50% of the cases. KBANN-12, learning rate= 0.5, momentum= 0.7 and seed= 7, classified 4 of 16 cases (25%) with an avg. pattern/error of 0.0712 (std= 0.0268) and 0.6280 (std= 0.3684) for training and testing respectively. Finally, KBANN-13, learning rate= 0.7, momentum= 0.5 and seed= 5, training and testing avg. pattern/errors 0.1228 (std= 0.1341) and 0.4729 (std= 0.3548) respectively, classified 7 (44%) of the cases. These results are presented in tables 12 and 13.

Training Phase		
	Performance	
Network	avg. pat/error	std
KBANN-10	0.1335	0.1379
KBANN-12	0.0712	0.0268
KBANN-13	0.1228	0.1341

Table 12: KBANN: *leave–one–out* method.

Testing Phase			
	Performance		
Network	average pat/error	std	Correct (%)
KBANN-10	0.4380	0.3617	8/16 (50%)
KBANN-12	0.6280	0.3684	4/16 (25%)
KBANN-13	0.4729	0.3548	7/16 (44%)

Table 13: KBANN: *leave–one–out* method

8 Discussion

Although the performance of both knowledge–free and knowledge–based neural networks was precluded by the complexity and scarcity of the data, it is clear how classification can be improved by adding some knowledge into a network. These results are consistent with the fact that a combination of learning techniques provides a more robust methodology than either technique alone. Comparisons between knowledge–free and knowledge–based networks presented in Table 11 emphasize how performance can improve with the addition of knowledge. In other words, the explanation–based module helps to overcome some of the difficulties during the learning process, because the incorporation of some knowledge into its topology provides the network with a more advantageous position at the beginning of the learning task. Also, the embedded expertise can guide the network, through the space of possible solutions, to reach a much better and accurate response to the presented patterns. Training time is also a point worth mentioning. Although it was not addressed in the results section, convergence timing was much shorter for knowledge–based than for knowledge–free networks. Finally, due to the scarcity and complexity of the available data, it was not possible to assess whether the performance of both types of networks was precluded either by the nature of the data or by the limited number of cases.

9 Conclusions

Complexity and scarcity are clearly two main constraints commonly found in real–life problems. Classification of *in vivo* ^{31}P MRS is not an exception. While demanding high levels of accuracy for any tool to be useful in a classification process, it also imposes tight limitations so difficult to overcome. Although neural networks are a flexible tool for classification, they require large training sets. This is an important limitation for such a connectionist learning technique. However, KBANN's combine the advantages of both connectionist and symbolic approaches to provide a more robust learning mechanism.

Both approaches are important for a knowledge–based network to succeed. The symbolic approach, represented by the 17 rule knowledge base proposed here, is responsible for guiding the network during the learning process. If knowledge is accurate and non–redundant, then a network will perform well. Otherwise, inaccurate knowledge could misguide the learning process and lead network away from convergence. Results presented in previous sections showed the advantages of a well-defined knowledge base. Even little knowledge from few rules can improve a network performance. As more knowledge was added into a network topology, its classification rate was augmented. This point is emphasized by three knowledge-based networks selected for further testing. All of them were selected from the 17 rule knowledge base batch. Thus, the importance of a well-defined knowledge base for a network to succeed was demonstrated.

Classification rates of 25–50% for the three knowledge–based networks during the testing phase,

stress the ability of KBANN's to generalize over previously unseen cases. Probably, these figures on their own do not accentuate the significance of the achieved results. However, it is necessary to bear in mind that due to the high demands imposed by the constraints of this classification problem, KBANN's performance appears precluded. Even more, these three KBANN's outperformed knowledge–free networks in spite of the fact that knowledge–free networks were trained using the whole 16 cases as training set and were not tested over unseen cases.

Additional data would be required to assess whether the relatively poor performance showed by KBANN's was due to either the complexity or scarcity of data. Although KBANN's methodology can cope with small data sets, it is possible that scarce and complex data such as the one used here, poses unbearable demands upon the network.

In summary, KBANN's are a reliable aid capable of classifying complex and limited data. KBANN's showed a better performance in classification of ^{31}P MRS of normal breast tissues when compared with knowledge–free networks. Results presented here are the first step towards a better understanding of cyclic metabolic and structural changes of mammary tissues. These experiments provide a sound base for further research into metabolic and structural alterations in cancerous cells of breast.

Acknowledgments

I would like to thank Consejo Nacional de Ciencia y Tecnología, México for their support and CRC Clinical Magnetic Resonance Research Group for their advice on magnetic resonance and metabolic changes in the normal breast, and for providing the data for these experiments. I also thank Gabriela Ochoa for her comments and advice on the final version of this document.

References

Drife, J. (1989). Breast modifications during menstrual cycle. *Int. J. Gynecol Obstet.*, *Suppl.*(1), 19–24.

Fanger, H., & Ree, H. (1974). Cyclic changes of human mammary gland epithelium in relation to the menstrual cycle–an ultrastructural study. *Cancer*, *34*(1), 574–585.

Ferguson, D., & Anderson, T. (1981). Morphological Evaluation of Cell Turnover in Relation to the Menstrual Cycle in the "Resting" Human Breast. *British Journal of Cancer*, *44*, 177–181.

Fox, J., Myers, C., Greaves, M., & Pegram, S. (1987). A systematic study of knowledge base refinement in the diagnosis of leukemia. In Kidd, A. L. (Ed.), *Knowledge Acquisition for Expert Systems. A Practical Handbook*, pp. 73–89. Plenum Press.

Johnson, L., & Johnson, N. (1987). Knowledge Elicitation Involving Teachback Interviewing. In Kidd, A. L. (Ed.), *Knowledge Acquisition for Expert Systems. A Practical Handbook*, pp. 73–89. Plenum Press.

Kidd, A. L. (1987). Knowledge Acquisition–An Introductory Framework. In Kidd, A. L. (Ed.), *Knowledge Acquisition for Expert Systems. A Practical Handbook*, pp. 1–16. Plenum Press.

Kuipers, B., & Kassier, J. (1987). Knowledge Acquisition by Analysis of Verbatim Protocols. In Kidd, A. L. (Ed.), *Knowledge Acquisition for Expert Systems. A Practical Handbook*, pp. 45–70. Plenum Press.

Longacre, T., & Bartow, S. (1986). A Correlative Morphologic Study of Human Breast and Endometrium in the Menstrual Cycle. *The American Journal of Surgical Pathology*, *10*(6), 382–393.

Merck (1997). The On-Line Merck Manual. http://www.merck.com.

Nazario, A., De Lima, G., Simoes, M., & Novo, N. (1995). Cell kinetics of the human mammary lobule during the proliferative and secretory phase of the menstrual cycle. *Bull Assoc Anat (Nancy)*, *79*(244), 23–27.

Nazario, A., Simoes, M., & De Lima, G. (1994). Morphological and ultrastructural aspects of the cyclical changes of the human mammary gland during the menstrual cycle. *Rev Paul Med*, *112*(2), 543–547.

Noordewier, M., Towell, G., & Shavlik, J. (1991). Training Knowledge-Based Neural Networks to Recognize Genes in DNA Sequences. *Advances in Neural Information Processing Systems*, *3*.

Olsson, H., Jernström, H., Alm, P., Kreipe, H., Ingvar, C., & *et al* (1996). Proliferation of the breast epithelium in relation to menstrual cycle phase, hormonal use, and reproductive factors. *Breast Cancer Research and Treatment*, *40*, 187–196.

Pandya, A., Chandwani, S., Das, T., & Pandya, K. (1995). Serum Calcium, Magnesium and Inorganic Phosphorous Levels During Various Phases of Menstrual Cycle. *Indian J. Physiol Pharmacol*, *39*(4), 411–414.

Payne, G., Dowsett, M., & Leach, M. (1994). Hormone–dependent metabolic changes in the normal breast monitored non–invasively by ^{31}P magnetic resonance (MR) spectroscopy. *The Breast*, *3*, 20–23.

Potten, C., Watson, R., Williams, G., Tickle, S., Roberts, S., Harris, M., & Howell, A. (1988). The effect of age and menstrual cycle upon proliferative activity of the normal human breast. *British Journal of Cancer*, *58*, 163–170.

Rumelhart, D., Hinton, G., & Williams, R. (1986a). Learning Representations by Back-propagating Errors. *Nature*, *323*, 533–536.

Rumelhart, D., McClelland, J., & *et al* (1986b). *Parallel Distributed Processing. Explorations in the Microstructure of Cognition*, Vol. 1: Foundations. MIT.

Söderqvist, G., Isaksson, E., von Schoultz, B., & *et.al* (1997). Proliferation of breast epithelial cells in healthy women during menstrual cycle. *American Journal of Obstetrics and Gynecology*, *176*(1 pt 1), 123–128.

Sordo, M. (1997). KBANN: Learning from small data sets. Tech. rep. 9701, School of Cognitive and Computing Sciences, University of Sussex.

Strnad, R., Vachoušek, J., Čech, E., Smejkal, V., & Velek, J. (1995). Physiology of the Breast. *Česka Gynekol*, *60*(2), 102–103.

Thrun, S. B. (1995). Lifelong Learning: A Case Study. Tech. rep., School of Computer Science, Carnegie Mellon University.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of Approximate Domain Theories by Knowledge-Based Neural Networks. In *VIII National Conference of Artificial Intelligence, AAAI*, Vol. 2, pp. 861–866.

Towell, G. G. (1991). *Symbolic Knowledge and Neural Networks: Insertion, Refinement and Extraction*. Ph.D. thesis, University of Wisconsin, Madison.

Twelves, C., Porter, D., Lowry, M., Dobbs, N., & *et.al.* (1994). Phosphorus–31 metabolism of post–menopausal breast cancer studied *in vivo* by magnetic resonance spectroscopy. *British Journal of Cancer*, *69*, 1151–1156.

Vogel, P., Georgiade, N., Fetter, B., Vogel, F., & McCarty, K. (1981). The Correlation of Histologic Changes in the Human Breast With the Menstrual Cycle. *American Journal of Pathology*, *104*, 23–34.

von Schoultz, B., Söderqvist, G., von Schoultz, E., & Skoog, L. (1996). Hormonal regulation of the normal breast. *Maturitas*, *23*(Supplement), 23–25.