Rewriting Logic Using Strategies for Neural Networks: An Implementation in Maude*

Gustavo Santos-García[1], Miguel Palomino[2], and Alberto Verdejo[2]

[1] Universidad de Salamanca
santos@usal.es
[2] Departamento de Sistemas Informáticos y Computación, UCM
miguelpt@sip.ucm.es, alberto@sip.ucm.es

Summary. A general neural network model for rewriting logic is proposed. This model, in the form of a feedforward multilayer net, is represented in rewriting logic along the lines of several models of parallelism and concurrency that have already been mapped into it. By combining both a right choice for the representation operations and the availability of strategies to guide the application of our rules, a new approach for the classical backpropagation learning algorithm is obtained. An example, the diagnosis of glaucoma by using campimetric fields and nerve fibres of the retina, is presented to illustrate the performance and applicability of the proposed model.

Keywords: Neural networks, rewriting logic, Maude, strategies, executability.

1 Introduction

Rewriting logic [8] is a logic of concurrent change that can naturally deal with states and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency. Indeed, rewriting logic was proposed as a unifying framework in which many models of concurrency could be represented, such as labeled transition systems, concurrent object-oriented programming, or CCS, to name a few [6, 9, 5].

Artificial neural networks [4] are another important model of parallel computation. In [7] it was argued that rewriting logic was also a convenient framework in which to embed neural nets, and a possible representation was sketched. However, and to the best of our knowledge, no concrete map has ever been constructed either following those ideas or any others. Our goal with this paper is to fill this gap. In our representation of neural networks we consider the evaluation of patterns by the network, as well as the training required to reach an optimal performance.

Since its conception, rewriting logic was proposed as the foundation of an efficient executable system called Maude [1]. Here we write our representation

* Research supported by Spanish project DESAFIOS TIN2006–15660–C02–01 and by Comunidad de Madrid program PROMESAS S–0505/TIC/0407.

J.M. Corchado et al. (Eds.): DCAI 2008, ASC 50, pp. 424–433, 2009.
springerlink.com

directly in Maude to be able to run our neural networks and apply them to a real case-study, the analysis of campimetric fields and nerve fibres of the retina for the diagnosis of glaucoma [3].

The paper is organized as follows. In Section 2 we review those aspects of Maude that will be used in our specification, mainly object-oriented modules and strategies. Section 3 introduces multilayer perceptrons and the backpropagation algorithm. Their specification in Maude, and an appropriate strategy for their evaluation and training, is presented in Section 4. The application of our implementation to the study of the diagnosis of glaucoma is considered in Section 5, and Section 6 concludes.

2 Maude

Maude [1] is a high performance language and system supporting both equational and rewriting logic computation for a wide range of applications. The key novelty of Maude is that besides efficiently supporting equational computation and algebraic specification it also supports rewriting logic computation. Mathematically, a rewrite rule has the form $l : t \longrightarrow t'$ *if Cond* with t, t' terms of the same type which may contain variables. Intuitively, a rule describes a local concurrent transition in a system: anywhere a substitution instance $\sigma(t)$ is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place.

Full Maude [1] is an extension of Maude with a *richer module algebra* of parameterized modules and module composition operations and with special syntax for object-oriented specifications. These object-oriented modules have been exploited in the specification of neural networks.

2.1 Object Oriented Modules

An *object* in a given state is represented as a term `< O : C | a1 : v1,..., an : vn >` where `O` is the object's name, belonging to a set `Oid` of object identifiers, `C` is its *class*, the `ai`'s are the names of the object's *attributes*, and the `vi`'s are their corresponding values. *Messages* are defined by the user for each application.

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms of associativity, commutativity, and identity) using rules that describe the effects of *communication events* between some objects and messages. The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The general form of such rules is

$$\begin{array}{l} M_1 \ldots M_n \;\; \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \;\; \longrightarrow \\ \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\ \quad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \;\; M'_1 \ldots M'_q \quad \textit{if Cond} \end{array}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and *Cond* is a rule condition. The result of applying

a rewrite rule is that the messages $M_1, \ldots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \ldots, Q_p$ are created; and new messages $M'_1, \ldots, M'_q$ are sent.

2.2 Maude's Strategy Language

Rewrite rules in rewriting logic need to be neither confluent nor terminating. This theoretical generality requires some control when the specifications become executable, because it must be ensured that the rewriting process does not go in undesired directions and eventually terminates. Maude's strategy language can be used to control how rules are applied to rewrite a term [2]. Strategies are defined in a separate module and are run from the prompt through special commands.

The simplest strategies are the constants `idle` and `fail`, which always succeeds and fails, respectively. The basic strategies consist of the application of a rule to a given term, and with the possibility of providing a substitution for the variables in the rule. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control how the rewrite conditions are solved by means of strategies. An operation `top` restricts the application of a rule just to the *top* of the term. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation (`;`), union (`|`), and iteration (`*` for 0 or more iterations, `+` for 1 or more, and `!` for a 'repeat until the end' iteration). Another strategy combinator is a typical 'if-then-else', but generalized so that the first argument is also a strategy. The language also provides a `matchrew` combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten. Recursion is also possible by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies.

For our implementation, the full expressive power of the strategy language will not be needed and all our strategies will be expressed as combinations of the application of certain rules (possibly instantiated), concatenation, and 'repeat until the end' iteration. For efficiency reasons, we have extended the previous strategy language with a new combinator `one`(S) which, when applied to a term t, returns one of the possible solutions of applying the strategy S to t.

3 Multilayer Perceptrons

A *neural network* is defined in mathematical terms as a graph with the following properties: (1) each node or *neuron* i is associated with a state variable x_i storing its current output; (2) each junction between two neurons i and k, called *synapse* or *link*, is associated with a real weight ω_{ik}; (3) a real *activation threshold* θ_i is associated with each neuron i; (4) a *transfer function* $f_i(y_k, \omega_{ik}, \theta_i)$ is defined for each neuron, and determines the activation degree of the neuron as a function

of its threshold, the weights of the input junctions and the outputs y_k of the neurons connected to its input synapses.

Multilayer perceptrons are networks with one or more layers of nodes between the layer of input units and the layer of output nodes. These hidden layers contain neurons which obtain their input from the previous layer and output their results to the next layer, to both of which they are fully-connected. Nodes within each layer are not connected and have the same transfer function. In our case, the transfer function has the form $f(\sum_k \omega_{ik} y_k - \theta_i)$, where $f(x)$ is a sigmoidal function. It is defined by $f(x) = 1/(1 + e^{(\nu - x)})$, which corresponds to a continuous and derivable generalization of the step function.

3.1 The Backpropagation Algorithm

The accuracy of the multilayer perceptron depends basically on the correct weights between nodes. The backpropagation training algorithm is an algorithm for adjusting those weights, which uses a gradient descent method to minimize the mean quadratic error between the actual outputs of the perceptron and the desired outputs.

Let x_{ij}^k and y_{ij}^k be the input and output, respectively, for the i pattern of node j of layer k. Let ω_{ij}^k be the weight of the connection of neuron j of layer k with neuron i of the previous layer. By definition of the perceptron by layers, the following relationships are fulfilled $x_{ij}^k = \sum_l \omega_{lj}^k y_{il}^{k-1}$; $y_{ij}^k = f(x_{ij}^k)$.

The mean quadratic error function between the real output of the perceptron and the desired output, for a particular pattern i, is defined as $E_i = \frac{1}{2} \sum_{j,k} (y_{ij}^k - d_{ij}^k)^2$, where d_{ij}^k is the desired output for pattern i of node j of layer k. In order to minimize the error function we use the descending gradient function, considering the error function E_p and the weight sequence $\omega_{ij}^k(t)$, started randomly at time $t = 0$, and adapted to successive discrete time intervals. We then have $\omega_{ij}^k(t+1) = \omega_{ij}^k(t) - \eta\, \partial E_l / \partial \omega_{ij}^k(t)$, where η is the so-called *learning rate constant*.

We can conclude that $w_{ij}(t+1) = w_{ij}(t) + \eta \delta_j x_i'$, where x_i' is the output of neuron i, and δ_j is an error term for node j. For output neurons, it must be $\delta_j = y_j(1 - y_j)(d_j - y_j)$. For a hidden node j, $\delta_j = x_j'(1 - x_j') \sum_k \delta_k w_{jk}$, where k ranges over all neurons in the layers above neuron j. Internal node thresholds are adapted in a similar manner.

4 Implementing the Multilayer Perceptron

In this section we focus on specifying a three-layer perceptron in Maude and designing a strategy for evaluation and training. In order to have a running net we need to specify the number of layers, the neurons in each of them, the weights of all links, and the input patterns which, in general, will be multiple. Whereas the object-oriented representation is very convenient for specifying their behavior, it is clear that introducing all these data directly in this form would be very cumbersome. Hence, we have decided to use matrices and vectors of values

to specify thresholds and weights, and to define equations and rules to transform them into the object-oriented representation.

The core of our representation of perceptrons in Maude revolves around the definition of two classes to represent neurons and synapses as individual objects:

```
class Neuron | x : Float, t : Float, st : Nat .
class Link | w : Float, st : Nat .
```

Each neuron object carries its current activation value `x`, depending on its threshold `t`, and an attribute `st` that will be used to determine whether the neuron has already fired or not, that is, whether it is still waiting for input or has already output a value. Similarly, link objects store their numerical weight and contain an attribute `st` to flag whether some value has already passed through them or not. A net then is a "soup" (a multiset) of neurons and links.

Neurons and links are identified by a name. We define two operations that take natural numbers as arguments and return an object identifier: for neurons, the numbers correspond to the layer and the position within the layer; for links, the numbers correspond to the output layer and the respective positions within each layer of the neurons connected.

```
op neuron : Nat Nat -> Oid .    op link : Nat Nat Nat -> Oid .
```

The evaluation of the network is essentially performed by repeated application of the rules `feedForward` and `sygmoid`. Rule `feedForward` calculates the weighted sum of the inputs to the neuron, whereas `sygmoid` just applies the sigmoidal function `syg` (defined somewhere else in the code) to the net input. As can be seen in `feedForward`, the attribute `st` of links is assumed to be 0 prior to their firing and becomes 1 once the information has been sent from one neuron to the other. Hence, pending some kind of reset, links can only be used once. Similarly, the rule `sygmoid` sets the attribute `st` of a neuron to 1 once the sigmoidal function has been applied.

```
rl [feedForward] : < neuron( L, I) : Neuron | x : X1 , st : 1 >
   <link(s L,I,J) : Link | w:W, st:0> <neuron(s L,J) : Neuron | x:X2, st:0>
=> <neuron(L,I) : Neuron | x:X1, st:1> <link(s L,I,J) : Link | w:W, st:1>
   < neuron(s L, J) : Neuron | x : (X2 + (X1 * W)) , st : 0 > .
  rl [sygmoid] :   < neuron(L, I) : Neuron |  x : X , t : T , st : 0 >
  =>   < neuron(L, I) : Neuron |  x : syg(_-_(X, T), L) ,  st : 1 > .
```

Evaluation of a perceptron starts by obtaining an input pattern through the rule `nextPattern`, which is guided by the message `netStatus`. A message of the form `netStatus(N0, 0, 0, N1)` means that the `s N0`-th pattern should be considered, and then the following patterns until the `N1`-th.

```
msg netStatus : Nat Nat Nat Nat -> Msg .
crl [nextPattern] :   netStatus(N, N1, N2, N0)  =>
netStatus(s N, N1, N2, N0) inPatternConversion(s N, inputPattern(s N), 0)
outPatternConversion(s N, outputPattern(s N), 0)   if  N < N0 .
```

Before starting the feedforward process, the values of the neurons in the input layer and the corresponding weights are reset. After that, the rule `introducePattern` inserts the input pattern in the neurons of the input layer and removes them from the configuration.

```
rl [introducePattern] : < neuron(0, I) : Neuron | x : X , st : 0 >
inputPattern(N, I, X0) =>  < neuron(0, I) : Neuron | x : X0 , st : 1 > .
```

Once we are done with the evaluation of all patterns, we compute the error and mark the current object `net(N)` as completed.

```
rl [computeError] :   < net(N0) : Net | e : E , st : 0 >
< neuron(2, I) : Neuron | x : X0 , st : 1 >  outputPattern(N, I, X1, 0)
=> < net(N0) : Net | e : (E + ((_-_(X1, X0)) * (_-_(X1, X0)))), st : 0 >
< neuron(2, I) : Neuron | x : X0, st : 1 >  outputPattern(N, I, X1, 1) .
```

4.1 Backpropagation in Maude

For training the net we need neurons and links to hold additional information, namely the error terms δ_j and the adjusted weights $\omega_{ij}^k(t+1)$. Since evaluation is part of backpropagation, we define `NeuronTR` and `LinkTR` as *subclasses* of `Neuron` and `Link` with an additional attribute to store the extra information.

```
class LinkTR   | w+1 : Float .   subclass LinkTR < Link .
class NeuronTR | dt : Float .    subclass NeuronTR < Neuron .
```

Note that the rules for evaluating a net also apply to these new objects; the new attributes are simply ignored. The next step demands the evaluation of the error terms before adjusting the weights. The calculation of these δ_j depends on whether we are working with the output or hidden layers. For the output layer, the corresponding rule is straightforward:

```
rl [delta2] :  outputPattern(N, I, D, 1)
< neuron(2, I) : NeuronTR | x : X , dt : DT , st : 2 >
=>  < neuron(2, I) : NeuronTR | x : X ,
        dt : (X * ((_-_(1.0, X)) * (_-_(D, X))) ) , st : 3 > .
```

The case for the remaining layers is a bit more involved and is split in three phases: the rule `delta1A` initializes `dt` to zero, `delta1B` below takes care of calculating the sum of the weights multiplied by the corresponding error term, and `delta1C` computes the final product. Again, in all these rules the status attribute `st` is correspondingly updated.

```
rl [delta1B] :  < neuron(1, J) : Neuron | dt : DT1, st : 2 >
< link(2, J, K):Link | w:W, st:2 > < neuron(2, K):Neuron | dt:DT2, st:2 >
=>  < neuron(1, J) : Neuron | dt : (DT1 + (DT2 * W)), st : 2 >
< link(2, J, K):Link | w:W, st:3 > < neuron(2, K):Neuron | dt:DT2, st:2 > .
```

Once the error terms are available, the updated weights can be calculated: rule `link1` does it for the hidden layer and `link2` for the output layer. Finally the old weights are replaced by the adjusted ones with the rule `switchLink`. Here we show rule `link1`:

```
rl [link1] :  < neuron(0, I) : Neuron | x : X1, st : 1 >
  < link(1, I, J) : Link  | w : W, w+1 : W1, st : 1 >
  < neuron(1, J) : Neuron | dt : DT, st : 3 >
=> < neuron(0, I) : Neuron | x : X1, st : 1  >
  < link(1, I, J) : Link | w : W, w+1 : (W + (eta * (DT * X1))), st : 3 >
  < neuron(1, J) : Neuron | dt : DT, st : 3 > .
```

4.2 Running the Perceptron: Evaluation and Training

Our specification is nondeterministic and not all of its possible executions may correspond to valid behaviors of a perceptron. Hence, in order to be able to use the specification to simulate the evaluation of patterns we need to control the order of application of the different rules by means of strategies.

The main strategy `feedForwardStrat` takes a natural number as argument and applies to a `Configuration` (that is, a perceptron), chooses a layer `L'` and applies rule `feedForward`, at random positions and as long as it is enabled, to compute the weighted sum of values associated to each neuron at the layer. When all sums have been calculated, it applies the sigmoidal function to all of them by means of rule `sygmoid` which, again, is applied at random positions and as long as it is enabled.

```
strat feedForwardStrat : Nat @ Configuration .
sd feedForwardStrat(L') := one(feedForward[L<-L'])!; one(sygmoid[L<-s L'])!.
```

There are two auxiliary strategies. The strategy `inputPatternStrat` takes care of making the successive patterns available and of resetting the appropriate attributes of the neurons and links, whereas `computeOutput` is invoked to compute the error once a pattern has been evaluated.

```
strat inputPatternStrat : @ Configuration .
sd inputPatternStrat :=
 one(resetNeuron) ! ; one(resetLink) ! ; one(nextPattern)  .
strat computeOutput : @ Configuration .
sd computeOutput :=  one(computeError) ! ; setNet  .
```

Last, all these previous strategies are combined into the evaluation strategy, which inputs the next pattern, computes the values of the neurons in the hidden and the output layers, and returns the error:

```
strat evaluateANN : @ Configuration .
sd evaluateANN :=  inputPatternStrat ; feedForwardStrat(0) ;
                   feedForwardStrat(1) ;  computeOutput  .
```

Then, to force the evaluation of the first `M` patterns by the multilayer perceptron the following command would be executed:

```
(srew ann netStatus(0, 0, 0, M) using one(evaluateANN) ! .)
```

where the input patterns would have been suitable defined and `ann` would be a term of the form:

```
neuronGeneration(0, input0, threshold0, 0)
neuronGeneration(1, input1, threshold1, 0) linkGeneration(1, link1, 0, 0)
neuronGeneration(2, input2, threshold2, 0) linkGeneration(2, link2, 0, 0)
```

Similarly as for evaluation, we need to define an appropriate strategy for training the perceptron. Assuming we have already calculated the output associated to a pattern, we next must calculate the error terms, use them to obtain the adjusted weights, and transfer them to the right attribute. That can be easily done by applying the rules defined in the previous section in the right order.

```
strat backpropagateANN : @ Configuration .
sd backpropagateANN :=  one(delta2) ! ; one(link2) ! ;
     one(delta1A) ! ; one(delta1B) ! ; one(delta1C) ! ;  one(link1) !  ;
     one(switchLink) !  .
```

Finally, training a net consists in evaluating a pattern with the strategy `evaluate ANN` and then adjusting the weights accordingly with `backpropagateANN`.

```
strat stratANN :  @ Configuration .
sd stratANN :=  evaluateANN ; backpropagateANN  .
```

5 Example: Diagnosis of Glaucoma

For the diagnosis of glaucoma, we proposed the use of a system that employs neural networks and integrates the analysis of the nerve fibers of the retina from the study with scanning laser polarimetry (NFAII;GDx), perimetry and clinical data [3]. In that work, the resulting multilayer perceptron was developed using MatLab.

We used the data from that project as a test bed for our specification of the backpropagation algorithm in Maude. Our results were equivalent and the success rate was of 100% but the execution time of our implementation lagged far behind, which motivated us to optimize our code. Since equations are executed much faster than rules by Maude and, in addition, do not give rise to branching but linear computations, easily handled by strategies, we simplified rules as much as possible. The technique used was the same in all cases and is illustrated here with the rule `feedForward`:

```
rl [feedForward] : C => feedForward(C) .
op feedForward : Configuration -> Configuration .
eq feedForward(C < neuron(  L, I) : Neuron | x : X1 , st : 1 >
        < link(s L, I, J) : Link  | w : W , st : 0 >
        < neuron(s L, J) : Neuron | x : X2 , st : 0 >)
=  feedForward(C < neuron(  L, I) : Neuron | >
        < link(s L, I, J) : Link  | w : W , st : 1 >
        < neuron(s L, J) : Neuron | x : (X2 + (X1 * W)) >) .
eq feedForward(C) = C [owise] .
```

The evaluation and training strategies had to be correspondingly modified since the combinator `!` was no longer needed. The resulting specification is obviously less natural, but more efficient.

6 Conclusions

We have presented a specification of multilayer perceptrons, in a two step fashion. First we have shown how to use rewrite rules guided by strategies to simulate the evaluation of patterns by a perceptron, and then we have enhanced the specification to make the training of the net possible. The evaluation process is straightforward, essentially amounting to the repeated application of two rules, `feedForward` and `sygmoid`, which further does credit to the suitability of rewriting logic as a framework for concurrency. The training algorithm requires more rules, but the strategy is also rather simple.

The simplicity of the resulting specification should be put in perspective. Our first attempts at specifying perceptrons made use of a vector representation like the one we have used here for inputting the data and similar to that proposed in [7]. Such representation was actually suitable for the evaluation of patterns but proved unmanageable when considering the training algorithm. The election of our concrete representation in which neurons and links are individual entities and which, at first sight, might not strike as the most appropriate, is of paramount importance.

In addition to the representation, availability of strategies turned out to be decisive. With the vector representation layers could be considered as a whole and there was no much room for nondeterminism, while the change to the object-oriented representation gave rise, as we have observed, to the possible interleaving of rules in an incorrect order. It then became essential the use of the strategy language to guide the rewriting process in the right direction.

As a result, our specification is the happy crossbreed of an object-oriented representation and the use of strategies: without the first the resulting specification would have been much more obscure, whereas without the availability of the strategy language, its interest would have been purely theoretical.

In addition to the novel application of rewriting logict to neural nets, the advantage provided by our approach lies on the allowing the subsequent use of the many tools developed in rewriting logic, such as the LTL model-checker or the inductive theorem prover [1, 5], to study the net.

The complete Maude code, the data used for the examples, and the results of the evaluation can be downloaded from `http://maude.sip.ucm.es/~miguelpt/`.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
2. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. In: Archer, M., de la Tour, T.B., Muñoz, C.A. (eds.) 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Part of FLOC 2006, Seattle, Washington, August 16. Electronic Notes in Theoretical Computer Science, vol. 174(11), pp. 3–25. Elsevier, Amsterdam (2007)

3. Hernández Galilea, E., Santos-García, G., Franco Suárez-Bárcena, I.: Identification of glaucoma stages with artificial neural networks using retinal nerve fibre layer analysis and visual field parameters. In: Corchado, E., Corchado, J.M., Abraham, A. (eds.) Innovations in Hybrid Intelligent Systems, Advances in Soft Computing, pp. 418–424. Springer, Heidelberg (2007)
4. Lippman, R.P.: An introduction to computing with neural nets. IEEE ASSP Magazine, 4–22 (1987)
5. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: Gabbay, D. (ed.) Handbook of Philosophical Logic, 2nd edn., vol. 9, pp. 1–81. Kluwer Academic Press, Dordrecht (2002)
6. Martí-Oliet, N., Meseguer, J.: Rewriting logic: Roadmap and bibliography. Theoretical Computer Science 285(2), 121–154 (2002)
7. Meseguer, J.: Research directions in rewriting logic. In: Berger, U., Schwichtenberg, H. (eds.) Computational Logic: Marktoberdorf, Germany, July 29 – August 6, vol. 165, pp. 347–398. NATO Advanced Study Institute (1997)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
9. Talcott, C.L.: An actor rewriting theory. In: Meseguer, J. (ed.) Workshop on Rewriting Logic and its Applications, WRLA 1996. Electronic Notes in Theoretical Computer Science, vol. 4, pp. 360–383. Elsevier, Amsterdam (1996)