

Towards the Verification and Validation of Online Learning Adaptive Systems*

Ali Mili
College of Computing Science
New Jersey Inst. of Technology
Newark, NJ 07102
mili@cis.njit.edu

Bojan Cukic, Yan Liu
Lane CSEE Department,
West Virginia University
Morgantown WV 26506-6109
{cukic,yanliu}@csee.wvu.edu

Rahma Ben Ayed
School of Engineering
University of Tunis II,
Belvedere 1002 Tunisia
rahma_k@yahoo.com

April 1, 2002

Abstract

Online Adaptive Systems in general, and learning neural nets in particular cannot be validated using traditional verification and validation techniques, because they evolve over time, and past learning data influences their behavior. In this paper we discuss a framework for reasoning about online adaptive systems, and see how this framework can be used to perform V&V on such systems.

Keywords

Verification and Validation, Formal Methods, Refinement Calculi, On-Line Learning, Neural Networks, Adaptive Control.

1 Introduction: Position of the Problem

1.1 On-Line Learning: An Emerging Paradigm

Adaptive Systems are systems whose function evolves over time, as they improve their performance through learning. The advantage of adaptive systems is that they can, through judicious learning, react to situations that were never individually identified and analyzed by the designer. If learning and adaptation are allowed to occur after the control system is deployed, the system is called *online adaptive system*.

Online adaptive systems are attracting increasing attention in application domains where autonomy is an important feature, or where it is virtually impossible to analyze ahead of time all the possible combinations of environmental conditions that may arise. The controlled processes (as well as the control law) are often non-linear and subject to noise, disturbances, time delays and other unmodeled dynamics. Therefore, it is more advantageous to learn the system's behavior, rather than attempt its precise functional description. Examples of autonomous control applications

*This work is funded by grants from NASA Dryden Flight Research Center, through the Institute of Software Research (Fairmont, WV), and from NASA Goddard Space Flight Center, through NASA IV&V Facility (Fairmont, WV).

are long term space missions where communication delays to ground stations are prohibitively long, and we have to depend on the systems' local capabilities to deal with unforeseen circumstances [14]. An example of the system dealing with complex environmental conditions are flight control systems, which deal with a wide range of parameters, and a wide range of environmental factors. These systems must maintain flight safety and criticality equivalent to traditional human piloted systems. Other proposed applications include collision avoidance systems, multi-vehicle cooperative control, intelligent scheduling in manufacturing [10], control systems for automobile steering based on feature recognition in images [9], etc.

In recent years several experiments evaluated adaptive computational paradigms (neural networks, AI planners) for providing fault tolerance capabilities in control systems following sensor and/or actuator faults [27, 28]. Experimental success suggests significant potential for future use. More recently, a family of neural networks, referred to as DCS (Dynamic Cell Structure) [15], have been used by NASA for on-line learning of aerodynamic derivatives [36] in a flight control system of an F-15. In the intelligent flight control system, the online neural learning DCS network provides the aircraft model's adaptation to the changes that may occur during the flight. The network is trained to the error in flight, i.e., the difference between the derivative values computed by a regression-based derivative estimator, and those provided by the preflight approximation algorithm (implemented by another neural network, which does not change in flight). The topology representing properties of the DCS network proved to be capable of providing the flight controller with the best available estimates of the aircraft's stability and control derivatives, while yielding a dramatically more compact way to store them. These advances were made possible by the fact that a DCS network eventually acquires ("learns") the connectivity structure, which represents the relation of topological proximity of points from the flight envelope.

The critical factor limiting wider use of neural networks and other soft-computing paradigms in process control applications, is our (in)ability to provide a theoretically sound and practical approach to their verification and validation. In the rest of the paper, we present a framework for reasoning about on-line learning systems in hope that it may become a candidate technology for their verification and validation.

1.2 Verifying On-Line Learning Systems

While they hold great technological promise, on-line learning systems pose serious problems in terms of verification and validation, especially when viewed against the background of the tough verification standards that arise in their predominant application domains (flight control, mission control). Adaptive systems are inherently difficult to verify/validate, *precisely because they are adaptive*. Specifically, consider that methods for software product verification are generally classified into three families [4]:

- *Fault Avoidance* methods, which are based on the premise that we can derive systems that are fault-free *by design*.
- *Fault Removal* methods, which concede that fault avoidance is unrealistic in practice, and are based on the premise that we can remove faults from systems after their design and implementation are complete.
- *Fault Tolerance* methods, which concede that neither fault avoidance nor fault removal are feasible in practice, and are based on the premise that we can take measures to ensure that residual faults do not cause failure.

Unfortunately, neither of these three methods is applicable *as-is* to adaptive systems, for the following reasons:

- *Fault Avoidance*. Formal design methods [13, 19, 26] are based on the premise that we can determine the functional properties of a system by the way we design it and implement it. While this holds for traditional systems, it does not hold for adaptive systems, since their design determines how they learn, but not what they will learn. In other words, the function computed by an online adaptive system depends not only on how the system is designed, but also on what data it has learned from.
- *Fault Removal: Verification*. Formal verification methods [1, 24, 25, 21] are all based on the premise that we can infer functional properties of a software product from an analysis of its source text. While this holds for traditional systems, it does not hold for adaptive systems, whose behavior is also determined by their learning history.

- *Fault Removal: Testing.* All testing techniques [11, 20, 23] are based on the premise that the systems of interest will duplicate under field usage the behavior that they have exhibited under test. While this is true for traditional deterministic systems, it is untrue for adaptive systems, since the behavior of these systems evolves over time. We have observed in [2] that adaptive systems fail to meet this requirement (of maintaining or enhancing their behavior) even when they *converge*.
- *Fault Tolerance.* Fault tolerance techniques [3, 31, 32, 35] are based on the premise that we have clear expectations about the functions of programs and programs parts, and use these expectations to design error detection and error recovery capabilities. With adaptive systems, it is not possible to formulate such expectations because the functions of programs/ program parts are not predetermined.

Because on-line learning systems are most often used in life-critical (e.g. flight control) and mission-critical (e.g. space) applications, they are subject to strict certification standards, leaving a wide technological gap —which we attempt to narrow (however slightly) in this paper. First, we survey existing approaches.

1.3 Existing Approaches

Traditional literature typically describes adaptive computational paradigms with respect to their use, as function approximators or data classification tools. In most cases, their “*correctness*” is measured in terms of a misclassification rate on specific data sets, or by their ability to interpolate and/or extrapolate between known function values. This evaluation paradigm may work well only for applications where the system learns on a “training set” and remains unchanged in operational usage. In an attempt to discuss verification and validation of neural networks, LiMin Fu [16] interprets verification to refer to correctness and interprets validation to refer to accuracy and efficiency. He establishes correctness by analyzing the process of designing the neural network, rather than the functional properties of the final product. An intuitively similar, but more elaborate approach has been described by Gerald Peterson [30]. Peterson describes the opportunities for verification and validation of neural networks in terms of the activities in their development life-cycle, as shown in Figure 1.

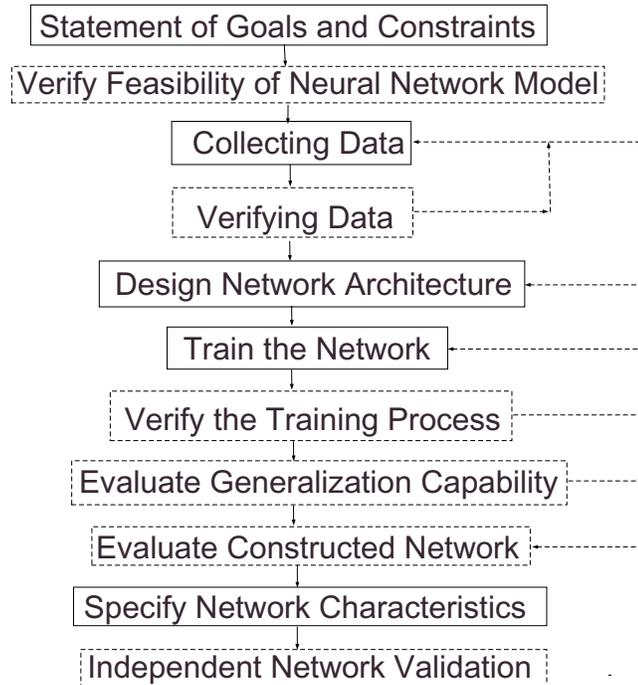


Figure 1: NN construction life-cycle.

If a problem is judged to be solvable by a neural network (feasibility phase), training data is gathered. Verification of the training data includes the analysis of appropriateness and comprehensiveness. This step is not fully applicable to on-line learning applications since training data are related to the real-time evolution of the system state, rather than the design choice. Verification of the training process typically examines the convergence properties of the learning algorithm in terms of achieving the desired optimal problem solution. Evaluation of interpolation and extrapolation capabilities of the network and domain specific verification activities set the stage for the overall verification and validation. The strong emphasis on domain specific knowledge, its formal representation and mathematical analysis is suggested in [18] too. Authors propose the analysis of the neural network with respect to conditions implying the existence of the solution (for function approximation) and the reachability of the solution from any possible initial state. Their third condition can be interpreted as condition for preservation of the learned information.

While meaningful and well organized, Peterson's approach provides little guidance on the choice of specific rigorous V&V techniques. Proposed techniques are mostly based on empirical evaluation through simulation and/or experimental testing. In an on-going effort, a group of researchers at NASA Ames Research Center are defining life-cycle V&V methods applicable to systems which have (an) integrated adaptive software component(s) [7]. In some cases, neural networks are modified to provide support for testing based (or on-line) validation of results. For example, Leonard et. al. [22] suggest a new architecture called *Validity Index*. A *Validity Index* network is a derivative of Radial Basis Function (RBF) network with the additional ability to calculate confidence intervals for its predictions based on the probability density of the "similar" training data observed in the past.

In a recent survey of methods for validating on-line learning neural networks, O. Raz [33] calls this approach *on-line monitoring and novelty detection* and attributes to it a significant potential for the future use. The other promising research direction, according to Raz, is periodic rule extraction from an on-line neural network and partial (incremental) re-verification of these rules using symbolic model checking. Practical hurdles associated with this approach include determining the frequency of rule extraction and impracticality of near real-time model checking of complex systems. LiMin Fu [16] discuss the verification and validation of neural nets, where he interprets verification to refer to correctness and interprets validation to refer to accuracy and efficiency. He establishes correctness by analyzing the process of designing the neural net, rather than the functional properties of the final product.

2 Tenets of a Refinement-Based Approach

2.1 Characterizing Our Approach

Our approach to the verification of on-line learning systems can be summarized in the following premises:

- We establish the correctness of the system, not by analyzing the process by which the system has been designed, but rather by analyzing the functional properties of the final product, and how these functional properties evolve through learning.
- Qualifying the first premise, we capture the functional properties of the system not by the exact function that the system defines at any stage in its learning process, but rather by a *functional envelope*, which captures the range of possible functions of the system for a given learning history. This concept will be more formally defined in section 3.1.
- In order to make testing meaningful, we need to ensure that the system evolves in a way that preserves or enhance its behavior under test. We call this *monotonic learning*, and we investigate it in some detail in section 4.1. Of course, on-line learning systems are supposed to get better as they acquire more learning data, but our definition of better is very specific: it means that the functional envelope of the system grows increasingly more refined with learning data (in the sense of refinement calculi [5, 8, 12, 17, 38]).
- In order to support some form of correctness verification, we must recognize that the variability of learning data and the focus on functional envelope (rather than precise function) weaken considerably the kinds of functional properties that can be established by correctness verification. Typically, all we can prove are minimal safety conditions; we refer to this as *safe learning* (proving that learning preserves safety conditions), and we discuss it in some detail in section 4.2.

In the sequel, we briefly introduce some mathematical background, which we use in the remainder of the paper.

2.2 Specification Structures

The verification and validation of systems, whether adaptive or not, can only be carried out with respect to predefined functional properties, which we capture in *specifications*. In this paper, we model specifications by means of binary relations. A *relation* R from set X to set Y is a subset of the Cartesian product $X \times Y$. A *homogeneous* relation on S is a relation from S to S . We use relations to represent specifications. Among relational constants we cite the identity relation, denoted by I , and the universal relation, denoted by L . Among operations on relations we cite the product, which we represent by $R \circ R'$ or by RR' (when no ambiguity arises), the complement, which we represent by \overline{R} , the inverse, which we represent by \widehat{R} , and the set theoretic operations of union and intersection.

We wish to introduce an ordering between (relational) specifications to the effect that a specification is greater than another specification if and only if it captures stronger functional requirements. We refer to this ordering as the *refinement ordering*, we denote it by $R \sqsupseteq R'$, and we define it as

$$RL \cap R'L \cap (R \cup R') = R'.$$

The following definition and proposition give the reader some intuition for the meaning of the refinement ordering.

Definition 1 A program P on space S is said to be correct with respect to specification R on S if and only if $\boxed{P} \sqsupseteq R$, where \boxed{P} is the function defined by program P .

Proposition 1 Specification R refines specification R' if and only if any program correct with respect to R is correct with respect to R' .

In [6], we have derived two propositions pertaining to the lattice properties of the refinement ordering. We present them here without proof, but with some discussion of their intuitive meaning.

Proposition 2 Two relations R and R' have a least upper bound (also called the join) with respect to the refinement ordering if and only if they satisfy the condition (called the consistency condition):

$$RL \cap R'L = (R \cap R')L.$$

When they do satisfy this condition, their join is denoted by $(R \sqcup R')$ and is defined by

$$R \sqcup R' = R \cap \overline{R'L} \cup R' \cap \overline{RL} \cup (R \cap R').$$

The consistency condition means that R and R' can be satisfied (refined) simultaneously. As for the expression of the join, suffice it to say that $(R \sqcup R')$ represents the specification that captures all the functional features of R (upper bound of R) and all the functional features of R' (upper bound of R') and nothing more (*least* upper bound). A crucial property of joins, for our purposes, is that an element A refines $R \sqcup R'$ if and only if it refines simultaneously R and R' . In other words, the join of R and R' represents the *sum* of all the functional features of R and R' . This sum can be derived only if R and R' do not contradict each other (re: the consistency condition). We argue in [6] that complex specifications can be structured in terms of simpler sub-specifications using the join operator.

In addition to discussing least upper bounds (joins), we also discuss greatest lower bounds (meets), which are introduced in the following proposition.

Proposition 3 Any two relations R and R' have a greatest lower bound (also called the meet), which is denoted by $(R \sqcap R')$ and defined by

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

The meet of R and R' is a specification that is refined by R (lower bound of R), refined by R' (lower bound of R'), and is maximal (*greatest* lower bound): in other words, it captures all the functional features that are common to R and R' .

The following lemma, which presents trivial lattice identities, will be generalized later for our purposes.

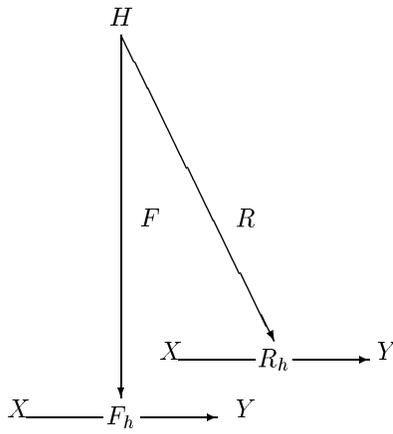


Figure 2: Abstract Computational Model

Lemma 1 *The following identities hold in any lattice:*

- $(A \sqsupseteq B) \vee (A \sqsupseteq C)$ logically implies $A \sqsupseteq (B \sqcap C)$.
- $(B \sqsupseteq A) \wedge (C \sqsupseteq A)$ logically implies $(B \sqcap C) \sqsupseteq A$.

The first clause stems readily from the transitivity of the refinement ordering, and the lattice identities $B \sqsupseteq (B \sqcap C)$, $C \sqsupseteq (B \sqcap C)$. The second clause can be proved by observing that the left hand side provides that A is a lower bound for B and C , hence it is refined by the greatest lower bound.

3 A Computational Model for On-Line Learning Systems

3.1 An Abstract Model

Before we discuss the specifics the verification methods we propose, we first introduce an abstract computational model for adaptive systems and their evolution through learning. Figure 2 depicts the abstract model we have of an online adaptive system; this model is purposefully generic, to support a wide range of possible implementations (RBF, DCS, MLP), and to enable us to focus on relevant computational features (as opposed to being distracted by implementation specific details). Our model includes the following features:

- Set X represents the set of inputs that may be submitted to the adaptive system.
- Set Y represents the set of outputs that the adaptive system may return as output.
- Set H represents the set of learning data histories that are submitted to the adaptive system for learning; typically, this set is nothing but the set of sequences of the form (x, y) , where $x \in X$ and $y \in Y$. We let ϵ represent the empty sequence (as an element of H).
- Function F is the function which, to each learning history h in H associates a function F_h from X to Y that captures the behavior of the adaptive system after receiving learning data h . According to this definition, the initial behavior of the adaptive system before any learning history is received is F_ϵ .
- Function R is the function which, to each learning history h in H associates a relation R_h from X to Y that captures the *learned behavior* of data h , and nothing else. Whereas F_h may include behavior that stems from its initialization, or stems from extrapolations, or stems from default options, R_h remains undefined or under-defined until learning data intervenes.

In order to elucidate the meaning of relation R_h , for history h , we consider the following development scenario for adaptive systems. An adaptive system is defined by some learning rule, which maps a learning history h into a function F_h ; the learning algorithm is also defined by means of implementation-specific parameters, including randomly chosen parameters. For the sake of abstraction, we denote the vector of implementation-specific parameters by a variable, say λ , and we let Λ be the set of possible values for λ . To fix our ideas, we can think of Λ as representing a family of possible implementations of the learning algorithm, and of λ as a specific implementation within the selected family; also, we denote by F_h^λ the function that captures the behavior of the adaptive system whose parameters vector is λ , upon receiving learning data h . With this background in mind, we let R_h be defined as follows:

$$R_h = \prod_{\lambda \in \Lambda} F_h^\lambda.$$

By virtue of the definition of meet, $\prod_{\lambda \in \Lambda} F_h^\lambda$ can be interpreted to represent the functional information that is common to all possible implementations of the learning algorithm, for all possible values of λ . While F_h^λ is dependent on λ , R_h is dependent on Λ .

As a corollary of this definition, consider the initial values of F_h^λ and R_h for $h = \epsilon$, i.e. at the beginning of the learning process. Whereas F_ϵ^λ represents the (mostly arbitrary) initialization of the function of the adaptive system, R_ϵ represents the information that all instances of F_ϵ^λ , for all values of λ in Λ have in common. In effect, R_ϵ captures all the functional information that stems from Λ , and that is specific to the family of learning algorithms being used.

The definition of R_h yields the following proposition, which we present without proof (the proof is a trivial lattice identity).

Proposition 4 *For all $\lambda \in \Lambda$, we have*

$$\forall h : F_h^\lambda \supseteq R_h.$$

This proposition stems readily from the definition of R_h as the meet of all F_h , for all h : the meet of many terms is lower than any one term.

3.2 A Concrete Model: The Back-Propagation Learning Algorithm

In this section, we consider the back-propagation learning algorithm, and we analyze it to show that it fits the abstract computational model that we have presented above. The back-propagation algorithm was first developed by Werbos in 1974 [37] but attracted little attention initially. It was later independently rediscovered by Parker [29] in 1982 and by Rumelhart, Hinton and Williams [34] in 1986. The version we present below, taken from [16], is due to [34].

- **Weight Initialization.** Set all weights and node thresholds to small random numbers. Note that the node threshold is the negative of the weight from the bias unit (whose activation level is fixed at 1).
- **Calculation of Activation.**
 1. The activation level of an input unit is determined by the instance presented to the network.
 2. The activation level O_j of a hidden and output unit is determined by

$$O_j = \sigma(\sum W_{ji} O_i - \theta_j),$$

where W_{ji} is the weight from an input O_i , θ_j is the node threshold, and σ is the sigmoid function:

$$\sigma(a) = \frac{1}{1 + e^{-a}}.$$

- **Weight Training.**
 1. Start at the output units and work backward to the hidden layers recursively. Adjust weights by

$$W_{ji}(t+1) = W_{ji}(t) + \Delta W_{ji},$$

where $W_{ji}(t)$ is the weight from unit i to unit j at time t and ΔW_{ji} is the weight adjustment.

2. The weight change is computed by

$$\Delta W_{ji} = \eta \delta_j O_i,$$

where η is a trial-independent learning rate ($0 < \eta < 1$) and δ_j is the error gradient at unit j . Convergence is sometimes faster by adding a momentum term:

$$W_{ji}(t+1) = W_{ji}(t) + \eta \delta_j O_i + \alpha (W_{ji}(t) - W_{ji}(t-1)),$$

where $0 < \alpha < 1$.

3. The error gradient is given by:

– For the output units:

$$\delta_j = O_j(1 - O_j)(T_j - O_j),$$

where T_j is the desired (target) output activation and O_j is the actual output activation at output unit j .

– For the hidden units:

$$\delta_j = O_j(1 - O_j) \sum_k \delta_k W_{kj},$$

where δ_k is the error gradient at unit k to which a connection points from hidden unit j .

4. Repeat iterations until convergence in terms of the selected error criterion. An iteration includes presenting an instance, calculating activations, and modifying weights.

We interpret this algorithm as defining function F_h (see section 3) by induction on the complexity (length) of h . If we recognize that F_h is not entirely determined by h but is also dependent on the arbitrary initial parameters (and their subsequent manipulations) then we rewrite this function as F_h^λ , where λ is the vector of weights

$$\lambda = \begin{pmatrix} \dots \\ \dots \\ W_{ji} \\ \dots \\ \dots \end{pmatrix}.$$

Also, we recognize that the range of values that weights can take evolves as the algorithm proceeds, hence the term Λ in the equations of section 3 should, in fact, be indexed with h ; to acknowledge this, we write it as Λ_h . Consequently, we find:

- Λ_ϵ , the initial set of possible weights, is defined by the *Weight Initialization* step in the back-propagation algorithm. The initial values of the weights are usually chosen rather small, since large weights cause the activation functions to saturate early, and cause the network to be stuck in a very flat plateau or a local minimum near the starting point. Typically, the initial values of weights are chosen as random values uniformly distributed between $\frac{-0.5}{FanIn}$ and $\frac{+0.5}{FanIn}$, where *FanIn* of a unit is the number of units which are fed forward into this unit [16].
- $\Lambda_{h.(x,y)}$ is obtained from Λ_h by applying the function detailed in the *Weight Training* step of the back-propagation algorithm. Specifically, if we let WT be the function detailed in this step, which has the form

$$\begin{pmatrix} W_{ji}(t+1) \\ \dots \\ \delta_j \end{pmatrix} = WT \begin{pmatrix} W_{ji}(t) \\ \dots \\ \delta_j \\ O_j \end{pmatrix},$$

then $\Lambda_{h.(x,y)}$ can be defined as follows:

$$\Lambda_{h.(x,y)} = \left\{ \left(\begin{array}{c} \dots \\ W_{ji}(t+1) \\ \dots \end{array} \right) \mid \left(\begin{array}{c} W_{ji}(t+1) \\ \dots \\ \delta_j \end{array} \right) = WT \left(\begin{array}{c} W_{ji}(t) \\ \dots \\ \delta_j \\ O_j \end{array} \right) \right. \\ \left. \wedge \left(\begin{array}{c} \dots \\ W_{ji}(t) \\ \dots \end{array} \right) \in \Lambda_h \right\}.$$

In light of this, we rewrite the characterization of R_h as follows:

$$R_h = \bigsqcap_{\lambda \in \Lambda_h} F_h^\lambda.$$

In particular, if we take $h = \epsilon$, we find that Λ_ϵ is the set of all admissible initial weights, and R_ϵ is the meet of all possible functions F_ϵ^λ for all admissible initial weights. Under some weak conditions (which are discussed in the sequel), we find a simple expression for R_ϵ :

$$R_\epsilon = \{(x, y) \mid \exists \lambda : (x, y) \in F_\epsilon^\lambda\}.$$

This formula is intuitively appealing: R_ϵ is the set of all input output pairs (x, y) such that (x, y) is in F_ϵ^λ for some admissible initial weighting λ . Note that while F_ϵ^λ reflects the arbitrary choice of an initial weighting, R_ϵ does not; it only reflects the learning algorithm and the specific network architecture. More generally, we intend R_h to reflect the learning algorithm, the network architecture, and the learning data—but *not to reflect any arbitrary choice of random weights*. Note also, on the expression above, that while F_h^λ is deterministic, R_ϵ is (very) non-deterministic; R_h is obtained from F_h^λ by abstracting away the arbitrary determinacy of F_h^λ .

In order to assess the variability of the system function with respect to the choice of initial weights, we have run an experiment on a simple back-propagation neural network with one hidden layer, and have submitted to it learning data about the *exclusive or* function. Also, we have selected the initial weights, and have observed how these affect the function F_h^λ for various values of h . Specifically, h is a sequence of *epochs*, where each epoch is made up of the four sets of inputs (combinations of two boolean variables) along with their corresponding outputs by the *exclusive or* function. The column labeled "10" in figure 3 represents the learning sequence h made up of ten epochs. By abuse of notation, we can represent h by the number of epochs in h . We can make the following observations:

- The initial weights have a large impact on the evolution of F_h^λ .
- This impact lasts well into the future, and does not completely disappear even after several thousand epochs.

For the purposes of our study, this means that R_h remains distinct from F_h^λ even for a long learning sequence h . Figure 3 also allows us to visualize the difference between F_h^λ and R_h : For example F_h^λ maps input (1,1) into 0.95718, whereas R_h also maps it into, among others,

$$0.70029, 0.50565, 0.51080.$$

4 Verification of on-Line Learning Systems

Given that we have derived the *functional envelope* of a an on-line learning system (as relation R_h), we discuss now how we can infer functional properties of the system. We discuss two methods in turn: *Monotonic Learning* and *Safe Learning*.

Initial Weights	Input	Iteration Times with Output						
		10	20	50	100	500	2000	Converge
W₀ =1.0								7285**
	(1,1)	0.95718	0.88763	0.64715	0.57213	0.50373	0.12861	0.04999388
	(1,0)	0.88929	0.75727	0.49946	0.48526	0.51578	0.88881	0.95669980
	(0,1)	0.88985	0.76131	0.50934	0.48949	0.51579	0.88868	0.95675480
	(0,0)	0.74329	0.58170	0.41602	0.45580	0.50102	0.09960	0.03909299
W₀=0.5								7560**
	(1,1)	0.70029	0.58756	0.53496	0.52377	0.51087	0.14863	0.04999296
	(1,0)	0.60074	0.51103	0.48290	0.48596	0.49385	0.87160	0.95670090
	(0,1)	0.60987	0.52137	0.48944	0.48869	0.49424	0.87141	0.95675580
	(0,0)	0.55051	0.49504	0.48686	0.49964	0.51760	0.11487	0.03909090
W₀ =0.0								8926**
	(1,1)	0.50565	0.50740	0.50880	0.50976	0.51116	0.50880	0.04999402
	(1,0)	0.48353	0.48525	0.48714	0.48836	0.48878	0.49985	0.95669870
	(0,1)	0.49364	0.49367	0.49203	0.49037	0.48888	0.50006	0.95675415
	(0,0)	0.51421	0.51434	0.51342	0.51227	0.51134	0.51613	0.03909125
W₀*								8942**
	(1,1)	0.51080	0.51098	0.51101	0.51096	0.51118	0.50909	0.04999226
	(1,0)	0.48304	0.48416	0.48627	0.48794	0.48876	0.49888	0.95670134
	(0,1)	0.49480	0.49397	0.49197	0.49027	0.48885	0.49911	0.95675653
	(0,0)	0.51010	0.51027	0.51051	0.51073	0.51137	0.51662	0.03908928

*: Random values range from -0.3 to $+0.3$.

** : Iteration times when network comes to convergence.

Figure 3: One Hidden Layer MLP NN for XOR Problem Trained by BP Algorithm with Different Initial Weights

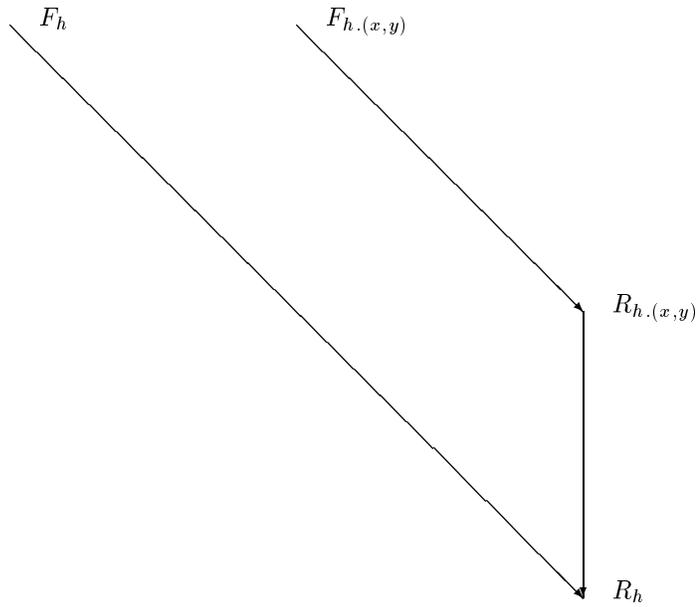


Figure 4: Monotonic Learning Increases R_h , Not Necessarily F_h .

4.1 Monotonic Learning

The idea of *monotonic learning* is to ensure that the adaptive system learns in a monotonic fashion, so that whatever claims we can make about the behavior of the system prior to its deployment are upheld while the system evolves through learning. Of course, we can hardly expect F_h to be monotonic with respect to h , since there is no way to discriminate between information of F_h that stems from learning and information that stems from arbitrary choices. In addition, whenever F_h^λ is total (which is fairly common), it is in fact maximal in the refinement ordering, hence cannot be further refined. We can, however, expect R_h to be monotonic, in the following sense.

Definition 2 *An adaptive system is said to exhibit monotonic learning if and only for all h in H , and for all (x, y) in $X \times Y$,*

$$R_{h.(x,y)} \sqsupseteq R_h$$

where $h.(x, y)$ is the sequence obtained by concatenating h with (x, y) .

Figure 4 illustrates in what sense monotonicity of R does not necessarily imply monotonicity of F . The challenge of this approach is to analyze what kinds of restrictions we must impose on the learning algorithm in order to ensure the monotonicity of R , or, alternatively, what kinds of learning algorithms ensure this property. Note that the refinement ordering is reflexive, hence nothing precludes us from a situation where $R_{h.(x,y)} = R_h$. One possible way to ensure monotonicity is to compare $R_{h.(x,y)}$ against R_h for refinement, and to discard (x, y) whenever the former does not strictly refine the latter. In practice this will work only if discarding learning data is an exceptional occurrence, rather than a routine occurrence.

The interest of monotonic learning is that whatever properties can be established by analyzing the adaptive system at any stage of its learning are sure to be preserved (in the sense of refinement) as the system learns. In particular, all the properties of R_ϵ (before learning starts) are maintained as the system learns. More significantly, any behavior that is exhibited at the testing phase is sure to be preserved (i.e. duplicated or refined) in field usage.

Traditional certification algorithms observe the behavior of a software product under test, and make probabilistic/statistical inferences on the operational attributes of the product (reliability, availability, etc). The crucial hypothesis on which these probabilistic/statistical arguments are based is that the software product will reproduce under field usage the behavior that it has exhibited under test. This hypothesis does not hold for adaptive neural nets, because they evolve their behavior (learn) as they practice their function. Of course, one may argue that they evolve their behavior

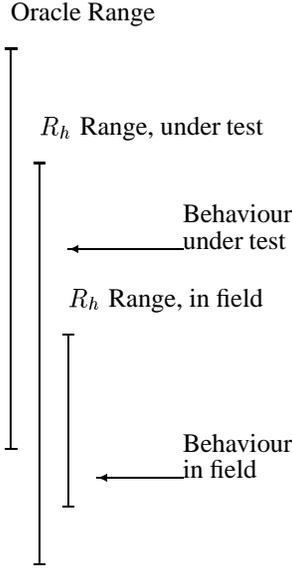


Figure 5: Convergence does Not Ensure Monotonicity

for the better; but *better* in the sense of a neural net (convergence, stability) is not necessarily better in the sense of correctness verification (monotonicity with respect to the refinement ordering). Concretely, a neural net may very well satisfy the test oracle in the testing phase, and fail to satisfy it in the field usage phase, even though it converges. See Figure 5.

In principle, to apply monotonic learning we need to derive a closed form expression of R_h , then we derive the condition provided in definition 2 and prove it. Because it is rather impractical to derive a closed form of R_h , this approach is unrealistic. As a substitute, we submit sufficient conditions for monotonic learning, starting with the following proposition.

Proposition 5 *If the following condition holds,*

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda \sqsupseteq F_h^{\lambda'},$$

then the pair (x, y) provides monotonic learning with respect to history h .

Proof. We must prove that under the condition cited above,

$$R_{h.(x,y)} \sqsupseteq R_h.$$

To this effect, we proceed by logical implications, starting from our hypothesis.

$$\begin{aligned}
& \forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda \sqsupseteq F_h^{\lambda'}, \\
\Rightarrow & \quad \{ \text{Definition of meet, transitivity} \} \\
& \forall \lambda : F_{h.(x,y)}^\lambda \sqsupseteq \bigcap \lambda' \in \Lambda F_{\lambda'} \\
\Rightarrow & \quad \{ \text{Definition} \} \\
& \forall \lambda : F_{h.(x,y)}^\lambda \sqsupseteq R_h \\
\Rightarrow & \quad \{ \text{Lattice Identity} \} \\
& (\bigcap \lambda F_{h.(x,y)}^\lambda) \sqsupseteq R_h \\
\Rightarrow & \quad \{ \text{Definition} \} \\
& R_{h.(x,y)} \sqsupseteq R_h.
\end{aligned}$$

We have found that often, function F_h^λ is total for all h and all λ ; this gives weight to the following proposition, which gives another (weaker, but no less general) sufficient condition of monotonic learning.

Proposition 6 *If F_h^λ is total for any history h and any initial weights λ , and the following condition holds,*

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda = F_h^{\lambda'},$$

then the pair (x, y) provides monotonic learning with respect to history h .

Proof. We consider the condition

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda \sqsupseteq F_h^{\lambda'}.$$

Because both terms of this inequation are function, this condition is equivalent to

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda \supseteq F_h^{\lambda'}.$$

Because both functions are total, this conditions can further be simplified as:

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda = F_h^{\lambda'}.$$

In other words, the learning pair (x, y) produces monotonic learning if and only if appending to learning history h produces the same outcome as starting with some other initial weight λ' and applying the learning history h . Presumably, λ' would have been a better initial weight than λ , since we get the same function for one less learning pair. This condition is not suggesting to choose a better λ , but rather is giving a sense to our concept of monotonic learning, which provides that as we learn more and more (i.e. as h increases in length), the range of possible values for function F_h^λ decreases. Note that there is no condition to the effect that every value of F_h^λ can be attained (by means of changing λ) for history $h.(x, y)$; hence the condition of corollary 5 is ensuring that the range of possible values for F_h^λ (which is the range of relation R_h) shrinks as h expands. We will discuss applications of this proposition in section 5.

4.2 Safe Learning

The main idea of *safe learning* is to ensure that as the adaptive system evolves through learning, it maintains some minimal safety property S . In other words, in addition to maintaining the identity

$$\forall h \forall \lambda, F_h^\lambda \sqsupseteq R_h,$$

which stems from the modeling of the system, we also require that the system maintains the following property

$$\forall h, F_h \sqsupseteq S$$

to ensure the safe operation of the adaptive system as it evolves through learning. By virtue of the lattice-like structure of the refinement ordering, we infer that F must satisfy:

$$\forall h \forall \lambda, F_h^\lambda \sqsupseteq (R_h \sqcup S).$$

See figure 6. This can be satisfied if and only if R_h and S do indeed have a join, i.e. if and only if they satisfy the consistency condition. The aggregate of conditions that characterize the safe learning of the adaptive system can be written as:

$$\forall h, R_h L \cap S L = (R_h \cap S) L$$

$$\forall h, F_h \sqsupseteq (R_h \sqcup S).$$

These conditions can be maintained by placing restrictions on the learning algorithms that can be deployed, or by controlling learning data that gets fed into the adaptive system, as per the following inductive argument:

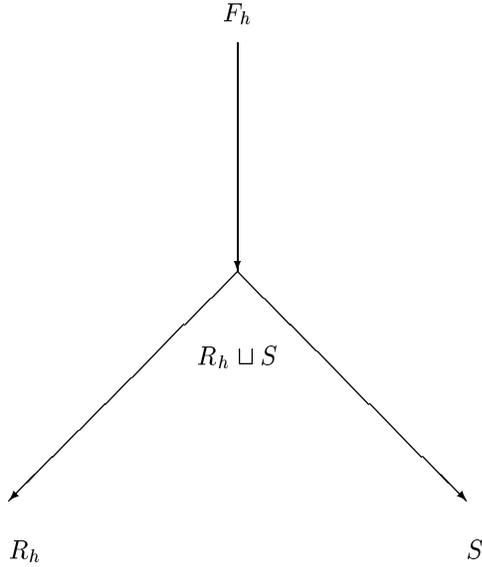


Figure 6: Safe Learning

1. As the basis of induction, these conditions hold for $h = \epsilon$, since R_ϵ is the minimal element of the lattice of refinement.
2. Given that they hold for h , we can ensure that they hold for $h.(x, y)$ by accepting entry (x, y) only if it does not violate these conditions.

4.3 Inductive Alternatives

Most traditional program verification methods tackle the complexity of the task at hand by doing induction on some dimension of program structure (control structure, data structure, depth of recursion, etc). Likewise, while the two methods we present here appear attractive, we have no doubt that they are complex in practice, because they rely on an explicit formulation of the functional envelope of the system. Hence we are focusing our attention on means to use induction in such a way that we can apply these methods without having to derive R_h . The key to the inductive approach is the ability to derive inductive relationships between R_h and $R_{h.(x,y)}$. In the case of the neural net we have discussed in section 3.2, we know the relation between successive weights (as defined by the *Weight Training* function, WT), the relation between a set of weights (λ) and the corresponding system function (F_h^λ) and we know how the functional envelope R_h is derived from system functions (by taking the meet for all values of λ). We must infer from this the relation between R_h and $R_{h.(x,y)}$. See figure 7.

5 Illustration: A Simple Back-propagation Network

We consider a simple Multi Layer Perceptron (MLP) with the simplest of architectures: one input layer, one hidden layer and one output layer, each containing a single neuron; see Figure 8. We want to use this example to discuss the condition of monotonicity; to this effect, we first write the expression of F_h^λ . We find,

$$F_h^\lambda = \{(x, y) | y = \sigma(w_2 \times \sigma(w_1 \times x))\}$$

where

- Function σ is defined by $\sigma(t) = \frac{1}{1+e^{-t}}$.

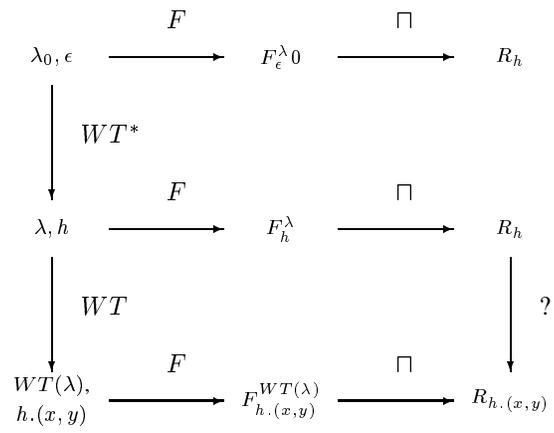


Figure 7: Inductive Structure

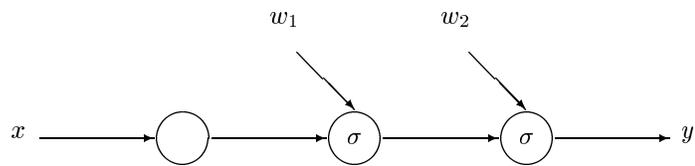


Figure 8: Architecture of a (Very) Simple Multi Layer Perceptron

- The vector $\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ obtained by backpropagation starting from initial weights λ , after the learning sequence h .

In order to articulate how the vector of weights $\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ is derived from the initial weights (λ) and from the learning history (h), we write

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = W_h(\lambda),$$

where function W_h is defined inductively (on h) by

- $W_\epsilon(\lambda) = \lambda$.
- $W_{h.(x,y)}(\lambda) = WT(W_h(\lambda), \begin{pmatrix} x \\ y \end{pmatrix})$,

where WT is, in turn, defined by

$$WT \left(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}, \begin{pmatrix} x \\ y \end{pmatrix} \right) = \begin{pmatrix} w_1 + \eta x \sigma(w_1 x) (1 - \sigma(w_1 x)) \sigma(w_2 \sigma(w_1 x)) (1 - \sigma(w_2 \sigma(w_1 x))) (y - \sigma(w_2 \sigma(w_1 x))) w_2 \\ w_2 + \eta \sigma(w_1 x) \sigma(w_2 \sigma(w_1 x)) (1 - \sigma(w_2 \sigma(w_1 x))) (y - \sigma(w_2 \sigma(w_1 x))) \end{pmatrix}.$$

where η is the learning rate.

By inspection of the formula of F_h^λ , we infer that F_h^λ is total (since σ is total), hence we use proposition 6, which provides the following sufficient condition for monotonicity:

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda = F_h^{\lambda'}.$$

We interpret this condition as:

$$\forall \lambda \exists \lambda' : (\forall t : F_{h.(x,y)}^\lambda(t) = F_h^{\lambda'}(t)).$$

Referring back to the formula of F_h^λ , we find that a sufficient (perhaps also necessary) condition of monotonicity is:

$$\forall \lambda, \exists \lambda' : W_{h.(x,y)}(\lambda) = W_h(\lambda').$$

In the sequel, we characterize cases under which this condition is satisfied; for each case, we present a brief argument, then discuss the significance of the case.

- *The first learning pair produces monotonicity.* If we take $h = \epsilon$, we find:

$$\begin{aligned} & \forall \lambda, \exists \lambda' : W_{h.(x,y)}(\lambda) = W_h(\lambda') \\ \Leftrightarrow & \quad \{ \text{Because } h = \epsilon \} \\ & \forall \lambda, \exists \lambda' : W_{h.(x,y)}(\lambda) = \lambda' \\ \Leftrightarrow & \quad \{ \text{By definition of } W_h \} \\ & \forall \lambda, \exists \lambda' : WT(W_h(\lambda), \begin{pmatrix} x \\ y \end{pmatrix}) = \lambda' \\ \Leftrightarrow & \quad \{ \text{Because } h = \epsilon \} \\ & \forall \lambda, \exists \lambda' : WT(\lambda, \begin{pmatrix} x \\ y \end{pmatrix}) = \lambda' \\ \Leftrightarrow & \quad \{ WT \text{ is a total function} \} \\ & \text{true.} \end{aligned}$$

Given that monotonicity means in effect that the new learning pair refines (enhances) prior knowledge, there is no doubt that the first pair always does (by contrast with subsequent pairs, which may conflict with prior knowledge).

- *Duplication produces monotonicity.* If we let h be a sequence of length 1, and let the new learning pair be a copy of the first pair, then we satisfy the condition of monotonicity. Formally,

$$\begin{aligned}
& \forall \lambda, \exists \lambda' : W_{h.(x,y)}(\lambda) = W_h(\lambda') \\
\Leftrightarrow & \quad \{ \text{By definition of } W_h \} \\
& \forall \lambda, \exists \lambda' : WT(W_h(\lambda), \begin{pmatrix} x \\ y \end{pmatrix}) = W_h(\lambda') \\
\Leftrightarrow & \quad \{ \text{Because } h = (x, y) = \epsilon.(x, y) \} \\
& \forall \lambda, \exists \lambda' : WT(W_h(\lambda), \begin{pmatrix} x \\ y \end{pmatrix}) = WT(W_\epsilon(\lambda'), \begin{pmatrix} x \\ y \end{pmatrix}) \\
\Leftrightarrow & \quad \{ \text{By definition of } W_h \} \\
& \forall \lambda, \exists \lambda' : WT(W_h(\lambda), \begin{pmatrix} x \\ y \end{pmatrix}) = WT(\lambda', \begin{pmatrix} x \\ y \end{pmatrix}) \\
\Leftarrow & \quad \{ \text{A sufficient condition} \} \\
& \forall \lambda, \exists \lambda' : \lambda' = W_h(\lambda) \\
\Leftrightarrow & \quad \{ W_h \text{ is a total function} \} \\
& \text{true.}
\end{aligned}$$

Repeating the same learning data does not create contradiction.

- *Convergence produces monotonicity.* We interpret convergence to be the situation where the new learning pair does not cause any change to the vector of weights. Formally,

$$WT \left(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}, \begin{pmatrix} x \\ y \end{pmatrix} \right) = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}.$$

Under this hypothesis, the condition of monotonicity

$$\forall \lambda \exists \lambda' : F_{h.(x,y)}^\lambda = F_h^{\lambda'}$$

holds vacuously for $\lambda' = \lambda$. The idempotence of WT holds in particular when the learning process has converged (for the submitted learning data). Also, the formula of WT for our sample example provides that we have idempotence whenever the learning pair (x, y) satisfies the conditions:

$$x = 0, y = \sigma\left(\frac{w_2}{2}\right).$$

6 Conclusion

On-line learning systems in general, and their neural net implementations in particular are gaining increasing acceptance in control applications, which are often characterized by complexity and criticality. A significant obstacle to their acceptance and usefulness/ usability is the lack of adequate verification/ certification methods and techniques, as all traditional methods and techniques are inapplicable. In this paper we are presenting a tentative computational model for on-line learning systems and we use this model to sketch verification methods. Among the main contributions of our work, we cite:

- An abstract computational model that captures the functional properties of an evolving adaptive system by abstracting away random factors in the function of the system, to focus exclusively on details that are relevant to the learning algorithm and the learning data.
- The integration of this computational model into a refinement logic, which establishes functional properties of adaptive systems using refinement-based reasoning.

- The introduction of two venues for verifying adaptive systems: one based on *monotonic learning* (the *adaptive* equivalent of testing), and one based on *safe learning* (the *adaptive* equivalent of proving).
- The introduction of a (sketchy, so far) framework for inductive reasoning on adaptive systems; this framework is based on the proposed computational model, and aims to support the *adaptive* equivalent of the inductive methods of program proving.
- Some preliminary exploration of monotonic learning, whereby we provide sufficient conditions for monotonic learning, discuss them, and illustrate them.

While this work is still in its infancy, we feel that it has introduced some meaningful concepts and has opened original venues for further exploration, by taking a refinement-based approach. We envisage the following extensions to this work:

- Experiment, be it on small examples, with the derivation of the functional envelope (R_h) of the system, and analyze what the conditions of monotonic learning and safe learning mean in practice. While it is easy to compute relation R_h extensionally, by listing some of its pairs (as we have done in figure 3), it is not trivial to derive a closed form expression of it.
- Investigate means to obviate the need to derive an explicit closed form expression for R_h , by exploring inductive arguments that allow us to ensure monotonic learning and safe learning without computing the functional envelope.
- Fine-tune the proposed computational model and investigate its applicability to other forms of on-line learning systems (other than the back-propagation algorithm).
- Derive tighter sufficient conditions for monotonicity, and further analyze the condition of safe learning.
- Explore inductive proof methods for adaptive systems, along the lines of the framework proposed in this paper.

This research is currently under way.

References

- [1] J.R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] Ch. Alexander, D. DelGobbo, V. Cortellessa, A. Mili, and M. Napolitano. Modeling the fault tolerant capability of a flight control system: An exercise in SCR specifications. In *Proceedings, Langley Formal Methods Conference*, Hampton, VA, June 2000.
- [3] H. Ammar, B. Cukic, C. Fuhrman, and Mili. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals of Software Engineering*, 10, 2000.
- [4] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, 11(12), December 1985.
- [5] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.
- [6] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.
- [7] M. A. Boyd, J. Schumann, G. Brat, D. Giannakopoulou, B. Cukic, and A. Mili. Ifcs project: Validation and verification (v&v) process guide for software and neural nets. Technical report, NASA Ames Research Center, September 2001.

- [8] Ch. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, New York, NY and Heidelberg, Germany, 1997.
- [9] M. Caudill. Driving solo. *AI Expert*, pages 26–30, September 1991.
- [10] C. H. Dagli, S. Lammers, and M. Vellanki. Intelligent scheduling in manufacturing using neural networks. *Journal of Neural Network Computing*, pages 4–10, 1991.
- [11] J. Dean. Timing the testing of cots software products. In *First International ICSE Workshop on Testing Distributed Component Based Systems*, Los Angeles, CA, May 1999.
- [12] Jules Desharnais, Ali Mili, and Thanh Tung Nguyen. Refinement and demonic semantics. In Brink et al. [8], chapter 11, pages 166–183.
- [13] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [14] D. Bernard et al. Final report on the remote agent experiment. In *NMP DS-1 Technology Validation Symposium*, Pasadena, CA, February 2000.
- [15] B. Fritzke. Growing self-organizing networks - why. In *European Symposium on Artificial Neural Networks*, pages 61–72, Brussels, Belgium, 1996.
- [16] LiMin Fu. *Neural Networks in Computer Intelligence*. McGraw Hill, 1994.
- [17] P. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.
- [18] D. Del Gobbo and B. Cukic. Validating on-line neural networks. Technical report, Lane Department of Computer Science and Electrical Engineering, West Virginia University, December 2001.
- [19] D. Gries. *The Science of programming*. Springer Verlag, 1981.
- [20] H. Hecht, M. Hecht, and D. Wallace. Toward more effective testing for high assurance systems. In *Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop*, Washington, D.C., August 1997.
- [21] Internet. Program verification system. Technical report, SRI International Computer Science Laboratory, 1997.
- [22] J. A. Leonard, M. A. Kramer, and L. H. Ungar. Using radial basis functions to approximate a function and its error bounds. *IEEE Transactions on Neural Networks*, 3(4):624–627, July 1991.
- [23] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, 13th IEEE International Conference on Automated Software Engineering*, pages 322–331, Honolulu, HI, October 1998. IEEE Computer Society.
- [24] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.
- [25] H.D. Mills, V.R. Basili, J.D. Gannon, and D.R. Hamlet. *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma, 1986.
- [26] C.C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.
- [27] M. Napolitano, G. Molinaro, M. Innocenti, and D. Martinelli. A complete hardware package for a fault tolerant flight control system using on-line learning neural networks. *IEEE Control Systems Technology*, January 1998.
- [28] M. Napolitano, C. D. Neppach, V. Casdorph, S. Naylor, M. Innocenti, and G Silvestri. A neural network-based scheme for sensor failure detection, identification and accomodation. *AIAA Journal of Control and Dynamics*, 18(6):1280–1286, 1995.

- [29] D.B. Parker. Learning logic. Technical Report S81-64, Stanford University, 1982.
- [30] G. E. Peterson. A foundation for neural network verification and validation. *SPIE Science of Artificial Neural Networks II*, 1966:196–207, 1993.
- [31] D. K. Pradhan. *Fault Tolerant Computing: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [32] B. Randall. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.
- [33] Orna Raz. Validation of online artificial neural networks —an informal classification of related approaches. Technical report, NASA Ames Research Center, Moffet Field, CA, 2000.
- [34] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume I: Foundations*. MIT Press, Cambridge, MA, 1986.
- [35] D.P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, Mass, 1982.
- [36] Being Staff. Intelligent flight control: Advanced concept program. Technical report, The Boeing Company, 1999.
- [37] P.J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. Technical report, Harvard University, 1974.
- [38] J. Von Wright. A lattice theoretical basis for program refinement. Technical report, Dept. of Computer Science, Åbo Akademi, Finland, 1990.