Practicing AI with the Portable AI Lab

Fabio Baj, Paolo Cattaneo, Mike Rosner

IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland
Tel.: +41 91 22.88.81, e-mail: paolo@idsia.ch

Abstract

The Portable AI Lab (PAIL) is an integrated collection of modules implementing established AI tools and techniques intended for use as a resource for teaching or learning artificial intelligence (AI). to well illustrated AI concepts by providing a set of incrementally complex demonstration programs, interactive tools to develop new examples, on-line context sensitive documentation with the description of the techniques involved, bibliographic references, and a set of exercises and projects. PAIL supplies a consistent interface that allows naive and more expert users to explore the core of AI algorithms in source form or graphically.

This paper describes and illustrates in depth the implementation and use of one of the twelve modules currently available and presents a discussion of the motivation behind, and some enhacement to, the communication between modules which is one of the key features of the system.

This public domain system is implemented in Common Lisp, CLOS, and Common Windows and runs on SPARC stations. It is distributed in source form via FTP. An alpha version of the same system is now available on Macintosh hardware.

1 Introduction

The Portable AI Lab is a computing environment containing a collection of state-of-the-art AI tools, examples, and documentation. It is aimed at those involved in AI courses (i.e. in both teaching and learning) at university level or equivalent. It has been developed under Swiss National Research Programme PNR 23 on AI and Robotics by IDSIA Lugano in collaboration with IFI, University of Zurich and the Laboratoire d'IA at the EPFL, Lausanne. The system is available free of charge.

The design of Portable AI Lab is motivated by the conviction that the acquisition of expertise in AI depends on extensive practical experience with a broad range of AI problems. Students should also appreciate that such problems are typically interdisciplinary in that they often involve more than one AI subdomain. The system has enough built-in functionality to enable its users to get such experience without having to build all the supporting tools from scratch and is intended to encourage the exploration of subdomains and the relationships between them.

There are a number of problems typically associated with the learning and teaching of AI which the system is designed to overcome. These include:

- There is no single, agreed definition of what AI is. Unlike such disciplines as physics or mathematics, it is difficult to identify a set of generally shared assumptions and goals from which the practice of AI can in some sense be said to follow deductively. Consequently, practice in AI tends to be associated with a set of very general problems and the use of a fairly heterogeneous set of tools and techniques to solve them. The Portable AI Lab attempts to provide a representative set of such tools and techniques.

- Although there is now no shortage of AI inspired software available at low or zero cost, it is difficult to find integrated collections of applications that have been designed from an educational perspective. This affects both the the grain size and functionality of applications, the use of visualisation techniques, the availability of help files, bibliographic references, and above all, the compatibility of different components.
- AI problems are typically interdisciplinary in nature, in the sense that they often involve techniques arising from a number of different subdomains. To see this we need only consider typical AI problems such as language understanding or knowledge acquisition, of which we shall have more to say in later sections of this paper. The acquisition of expertise in AI problem-solving therefore involves becoming familiar with both the components of interdisciplinary problems and the relationships between their parts.

The functionality is provided though a number of modules all implemented in Common Lisp and CLOS concerned with:

- Automatic Theorem Proving
- Natural Language Processing
- Rule-Based Inference
- Truth Maintenance Systems
- Constraint Satisfaction Problems
- Learning
- Knowledge Acquisition
- Genetic Algorithms
- Neural Networks

Each module comprises a fully implemememented and documented computational kernel over the subdomain in question, together with a representative set of autonomous demonstrations and running examples. Unlike many similar systems, a lot of attention has been paid to providing appropriate graphic visualisations for key algorithms and concepts. In addition, sufficient documentation is included to enable the user to retrieve key literature references and to understand the architecture and specifics of the implementation.

In the remainder of the paper we will address two topics. First, we describe the implementation of a typical module to illustrate its underlying structure and to point out the advantages to the end user. The second topic is intermodule communication, one of the key features of the system. That section of the paper discusses the motivation and the mechanism used to provide intermodule communication. We conclude by suggesting some future directions for intermodule communication in order to open the discussion on how to develop a powerful system for exploring interdisciplinary AI applications.

2 Designing a module: Constraint Satisfaction

This section illustrates some issues involved with the design and the implementation of a typical PAIL module, using that concerned with Constraint Satisfaction Problems (CSPs) as an example. The understanding of a concept often involves the process of seeing the relation between its many different (but in some sense isomorphic) representations. In some cases "understanding" is exactly the recognition of a mapping between different representation spaces. For this reason the designer of a module must be careful in defining all these levels of description in a way that suggests to the student the existing links. These design choices should ultimately be reflected in the CLOS coding style: the Lisp program (or, at least segments of it) have been expressly thought as fundamental parts of the system documentation. There has been a long debate during the development of the PAIL system about whether or not to provide users with direct access the Lisp code: the problem is that the system is intended to satisfy a broad set of users ranging from novices to expert Lisp programmers. For the former class of users it is important to provide graphics, on-line, context-sensitive help, and demonstrations: there is no need to show any language level implementation issue. On the other hand, expert users may feel "imprisoned" if they can not put their hands inside the system: a door to Lisp seems to provide the best solution.

2.1 Choosing the right algorithm

The basic step in the design of a tutorial module is the choice of a particular technique/algorithm to be implemented. In general a candidate algorithm should have the following characteristics: it should be well-known and accepted in the basic AI literature, understandable, as similar as possible to both the theory in the relevant field and to the code used to implement it. In the case of Symbolic Constraint Propagation we decided on Mackworth's AC3 [1], which is a generalization of the Waltz's well-known filtering algorithm [9], created for the domain f interpretation over three dimensions of two-dimensional drawings. The class of constraint satisfaction problems considered in this module are those in which there is a set of variables each to be instantiated in an associated domain and a set of boolean constraints limiting the set of possible values for specified subsets of the variables. A surprisingly large number of apparently different applications can be formalized in this way. Some of them are: map-coloring problems, crypto-arithmetic, geometric puzzles, crossword puzzles, n-queens problems, graph homomorphisms and isomorphisms, spatial layout, edge detection in computational vision, etc. Examples of such algorithms written in Lisp can be found in [6], and some of the core algorithms are explicitly taken from this source: however it is worth noting that PAIL designers had the additional problem of integrating standard Lisp algorithms with the code for graphical visualization, possibly avoiding confusion in the reader. We found very useful the adoption of an object-oriented programming style in order to encapsulate implementation details and let the user focus on the algorithms. In the PAIL system efficiency is often sacrificed in favour of clarity of code. Here are examples of class definitions and algorithms for the module of constraint propagation:

```
(defclass constraint-network ()
  ((nodes :initarg :nodes :accessor nodes :initform nil)
   (main-net :accessor main-net :initarg :main-net :initform nil)))

(defclass node ()
  ((name       :initarg :name     :accessor name     :initform nil)
   (domain     :initarg :domain     :accessor domain :initform nil)
   (neighbors  :initarg :neighbors  :accessor neighbors :initform nil)
   (possible-values :initarg :possible-values  :accessor possible-values :initform nil)))

(defmethod propagate-constraints ((net constraint-network))
```

```
  (loop (if (empty-queue) (return net))
(let* ((arc (dequeue)) (node1 (first arc)) (node2 (second arc))
       (old-num (length (possible-values node1))))
  (show-arc arc)
  (arc-consistency node1 node2)
  (if (impossible-p node1) (return nil))
  (when (< (length (possible-values node1)) old-num)
    (notify-reduction node1 node2 old-num)
    (enqueue node1)))))
```

The function propagate-constraints implements part of the classical algorithm for CSP quite readably and consistently with standard literature: however its execution must be massively interlaced with hidden code for graphical representation and animation, debugging and stepping: for instance, from the perspective of the above function an arc is simply a pair of nodes, but each node knows how to display and highlight itself when the function show-arc is invoked. Object-orientation allows to hide this low-level details from the program reader.

2.2 Smoothing the learning curve: autonomous demonstrations

There are often AI techniques which require of the user considerable initial efforts to enter in a particular domain, so that s/he can find answers to fundamental questions like: What is this technique for? Which problems can I solve with it? What kind of representation does it use? How does it work? How can I use this tool? Every module therefore includes a set of autonomous demonstrations illustrating the basic features of the technique and the tool. These play a crucial role for a module to be accepted and used by students. Since the user of a demonstration might be completely ignorant in the subject, these are normally based on domains with clear commonsense interpretations (figure 1 shows the CSP module in demo mode).

2.3 Providing a wide set of runnable examples

For every module, much attention has been devoted to the preparation of directly runnable examples in order to provide the widest view on the range of applicability of the technique. They are generally taken from the standard literature and provide additional information about the concepts to be acquired. Running examples also provide a useful guide to the user wishing to write his own applications. In the case of a CSP a running example is a file containing Lisp expressions defining the network structure together with the unary and binary constraints.

The definitions in the previous paragraph suggest that the CSP module may require the user to define the following functions:

```
(defun P-1 (var value) <BODY>)
```

this function implements the set of unary predicates : it must return T or NIL

```
(defun P-2 (var1 value1 var2 value2) <BODY>)
```

this function implements the set of binary predicates and it must return T or NIL

```
(defun VARIABLE-DOMAIN (domain-name) <BODY>)
```

this function must return the contents of the given domain name. It is used to initially fill the associated domains: it must return a list of values.

The choice of using Lisp as an input language might cause problems to the PAIL system when the user makes syntactic or semantic errors: for this reason all the errors occurring during the load of a CSP

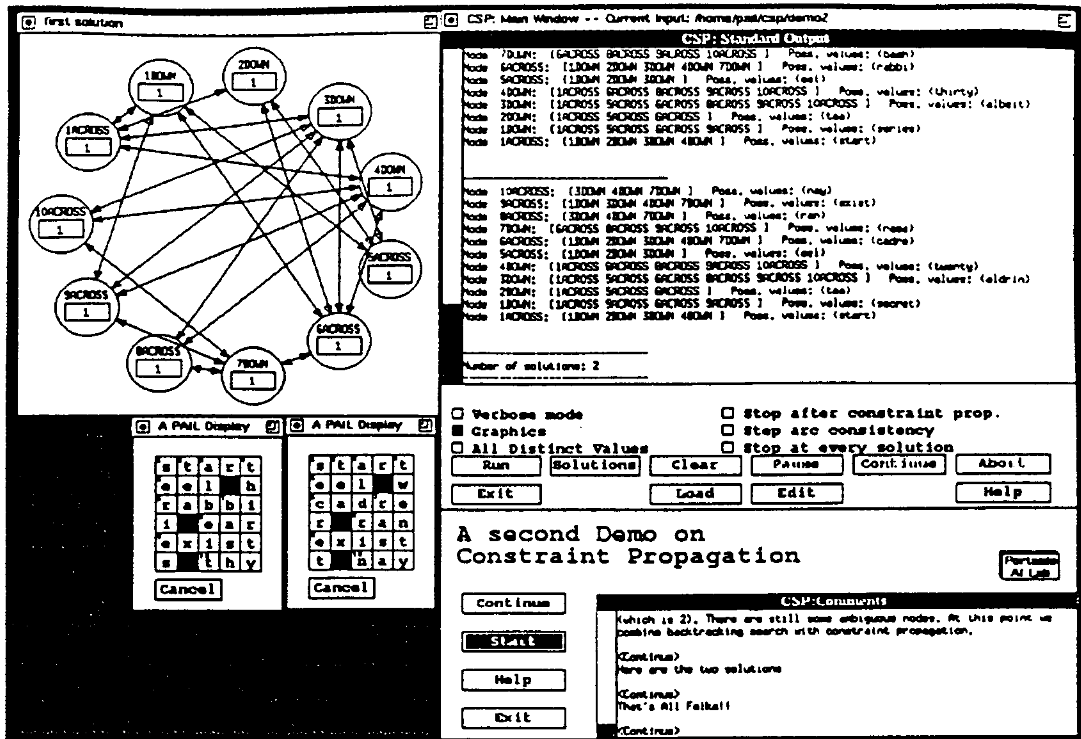

Figure 1: The Constraint Propagation Module in demo mode

description file and during the evaluation of user-defined functions are caught and the appropriate debug information is displayed on the standard output window. The following is an example of an input file describing a simple problem of map colouring. Notice the command **construct-network** with which the user defines a constraint network.

```
(defun p-1 (var value) t)

(defun p-2 (var1 v1 var2 v2)(not (equal v1 v2) ))

(defun variable-domain (domain-name)
  '(red green blue black))

(construct-network '((r1 D r2 r3 r5 r6) (r2 D r1 r3 r4 r5 r6)
     (r3 D r1 r2 r4 r6) (r4 D r2 r3)
     (r5 D r6 r1 r2) (r6 D r3 r5 r2 r1)))
 ;;optional part:
 (setq *all-distinct-values* nil)
 (setq *stop-at-sol* t)
```

2.4 Selecting and presenting text-based tutorial sources

Text-based sources of documentation (both online and paper) form an important component of every module, casting concepts acquired by running the system into a more formal framework that is consistent with the standard literature. The following formalization is an example of the kind of text-based information which can be found on-line within the Portable AI Lab.

A constraint satisfaction problem (CSP) is characterized as follows: given is a set V of variables $v_1, ..v_n$. Associated with each variable v_i is a finite domain D_i of possible values for variable v_i. We define a set of unary constraints $P_1, P_2, .., P_n$ and a set of binary constraints $P_{12}, P_{13}, .., P_{ij}, .., P_{kn}$ In general we may represent the satisfiability decision problem for a CSP as equivalent to determining the truth value of a well-formed formula in first order logic: $(\exists x_1)(\exists x_2)\ldots(\exists x_n)(x_1 \in D_1)(x_2 \in D_2)\ldots(x_n \in D_n)P_1(x_1) \wedge P_2(x_2) \wedge \ldots \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \wedge \ldots \wedge P_{(n-1)n}(x_{n-1}, x_n)$

2.5 Choosing a method to represent algorithms graphically

The formal specification of problems (see for instance the logic-based specification of CSPs in section 2.4) does not necessarly suggest effective algorithms for their solution, although it does provide concise ways to specify input to the system. Furthermore there is a gap between an abstract definition for a given algorithm and its implementation in a particular programming language. In general a module employs graphic tools to present algorithms and data structures: examples are decision trees (inductive learning) spreadsheets (knowledge acquisition, learning), proof trees (explanation-based-generalisation, truth maintenance, theorem proving, rule-based systems), networks (augmented transistion networks, constraint satisfaction).

Mackworth's AC3 algorithm represents a CSP as a network consisting of a graph with a vertex for each variable v_i and its associated domain D_i, and an edge between the vertices corresponding to each pair of directly constrained variables (i.e pairs of variables v_i, v_j for which the binary predicate P_{ij} is defined). Most textbooks introduce the AC3 algorithm using pictures representing constraint networks before giving a formal specification. The obvious choice was to provide the user with a conceptual view of the Constraint Propagation algorithm by animating on the screen this kind of picture (see figure 1). The possibility of interacting with these graphics adds to the system functionalities and is useful when debugging user-defined examples. Again, object orientation helps to write code which separates conceptual data with graphic-oriented features. The class **node-with-graphics** inherits the attributes of a generic node of a constraint network, and specializes it to have a graphic representation on the screen. The **domain-button** slot is a push-button with which the user can inspect interactively the domain of the node (see figure 1). Nonetheless the algorithms for constraint propagation still work on the more abstract object **node**

```
(defclass node-with-graphics (node)
   ( xpos    :initarg  :xpos :accessor xpos   :initform nil)
   ( ypos    :initarg  :ypos :accessor ypos   :initform nil)
   ( radpos :initarg  :radpos :accessor radpos  :initform nil)
   ( domain-button :initarg :domain-button :accessor domain-button  :initform nil)))
```

2.6 Linking the general algorithm with specific problem-instances

We saw that CSPs can be represented and solved with a constraint network. But we should also teach the student how to match apparently different problems like crossword puzzles, computer vision or map colouring in the same unifying framework. In general, a useful aid on this side is to provide, alongside the abstract view of algorithms, pictures showing the problems in their original representation space. The process of comparing, on the same screen, a picture with, for instance, a real crossword puzzle and the corresponding network seems to be very useful for students to acquire the ability to represent and solve new problems. If the user wants to customize the display of solutions according to a particular example (see for instance crosswords and puzzles) he/she can define a optional function named

```
(defmethod DOMAIN-DEPENDENT-SHOW-SOLUTION ((c-n constraint-network) <BODY>)
```

which will be called whenever a solution is displayed on the standard ouput. The variable **c-n** and its accessor functions are defined and described in the public source code. To issue a message on the CSP Standard Output the user can use the function **(message (string))**

2.7 Providing advanced examples

If we claim that the peculiarity of Constraint Propagation algorithms is their efficiency when dealing with huge search spaces, we must provide complex examples exploiting this feature. A full-page crossword puzzle, with about 10^{305} possible solutions is a good candidate: it would require thousands of years with a generate-and-test approach but the student can solve it in few minutes using Constraint Propagation. In general every module, besides classical toy examples, supplies harder applications, eventually giving the student some hints on the possible limitations of a particular technique. A module must also provide easy and flexible ways for the student to set up new experiments: this influences the design of the input-output interface. When an advanced user understands how the basic algorithms work, he will eventually be interested in re-using pieces of CLOS code for research or development. This is supported through fully documented and modular source code (the Common Lisp Package System was fundamental in this context). Furthermore the module designer must provide a framework for sharing information with the other modules, as shown in the next sections.

3 Intermodule communication

The previous section described a typical module in the Portable AI Lab. All other modules have been developed and built following a similar style and guidelines, one of the goals of which is to provide and facilitate communication between modules.

3.1 Motivations behind intermodule communication

Artificial Intelligence problems are typically interdisciplinary. A program with the goal of teaching AI techniques must address the problem of providing methods to explore interaction among different algorithms.

We can exemplify a few typical AI domains where interaction between different techniques seems fundamental:

- **Logic and language**: a cluster based on the modules for Augmented Transition Networks (ATN), Chart Parsing (CKY), and Theorem Proving. This cluster applies to the problem of establishing the semantic consistency of English (or any other language) sentences.

- **Machine learning and knowledge acquisition**: a typical application may be set up as follows: knowledge from a human expert can be elicited, refined and structured in tables using the Repertory Grid module [8]. Tables can be automatically converted in sets of examples to be given as input for the inductive learning module ID3 [7], which, in turn, is able to generate decision trees summarizing the original knowledge. This representation is also useful for evaluating the accuracy of the knowledge structures elicited by the Repertory Grid module.

- **Rule-based systems:** this cluster is centered around the rule-set data type. Rules can be used to perform backward or forward reasoning, eventually producing explanation trees which can be given as input to an explanation based learning algorithm as described in [5] to produce more specialized rules. Proof trees and rules are also the basic data structure for truth maintenance system [3]. Notice that knowledge acquired as decision trees by means of the Machine Learning cluster can be transformed into rule sets and integrated with this subset of modules.

The possibility of implementing some of these connections, and possibly discovering new ones, has always had a high priority on the list of activities associated with the development of the Portable AI Lab.

3.2 A mechanism of intermodule communication: the Pool

Individual modules can be regarded as units that consume one or more input items and produce one or more output items. Perhaps the simplest subclass of interdisciplinary applications are those for which the output of one module is used as the input to another as in the case of Logic and Language where the output of ATN parse, a translation from an English sentence to a first order logic formula, is passed directly to the Automated Theorem Prover module. A slightly more complex class of applications arises when modules have multiple inputs and outputs belonging to different types. For example, an ATN requires a grammar, a lexicon, and of course, a sentence in order to produce some output.

The Pool (see Allemang [2]), a data oriented approach to intermodule communication, was created in the first instance to make this kind of communication possible and in addition:

- to insulate modules (and their authors) from the peculiarities of any particular kind of file system implementation.
- to minimise the knowledge required by one module concerning the exact input or output formats used by other modules.

To achieve this the Pool is a logical structure similar to a file system that lies between the modules and the physical file system. All communication between modules and the Pool is assumed to be in terms of typed objects that are implemented as CLOS classes. The Pool has access to algorithms that permit conversions between some (not all) pairs of types. This permits automatic conversions between types wherever possible, so that for example, an object of type decision tree can be converted to one of type rule set if a particular module required it. Each module must therefore define a strongly typed interface to the Pool.

The conversion mechanism in Pool takes advantage of the CLOS properties. Every object must be loaded through the Pool which passes the data on file through the proper *change-type* method to provide conversion and then to the module. The default behavior is when no conversion is needed (i.e. we are loading the original data structure into the module). In this case the conversion method look pretty simple:

```
(defmethod change-type ((a op-set) (b op-set)) a)
```

If a conversion is possible between a pair of modules the programmer needs to write a method *change-type* to convert the data object accordingly.

```
(defmethod change-type ((dtree decision-tree) (rset rule-set))
  (make-instance 'rule-set :name-part "Rule-set" :rule-set-part (mapcar #'make-rule-from-path
        (intern-all (get-paths dtree) :dump))))

(defun make-rule-from-path (path)
  (make-instance 'rule :name-part "ID3-Rule" :if-part (do* ((l path (cddr l))
     (result nil (cons (list (second l) (first l)) result)))
   ((null (cddr l)) result))
 :then-part (list (second path) (first path))))
```

For all other pairs of incompatible modules the default *change-type* method will be invoked which just returns a warning.

```
(defmethod change-type (a b)
  (documentation-print (format nil "Conversion not possible from type ~a to ~a."
       class-name (class-of a)) (class-name (class-of b))) nil)
```

The Pool itself is of course implemented on the host file system: any number of Pool objects that work together can be thus be stored in a single file.

3.3 The Pool to the user

The user has two different views on the Pool. The first, accessible from the main window of each module, allows the user to "load" objects from the file system into the Pool and to edit the contents of the Pool. The following operations are permitted:

Get File: select a file from the current directory and load the objects it contains into the Pool.

Maintain File: add or delete objects from a file.

Maintain Pool: delete objects from Pool.

The second view is accessible from input or output buttons provide facilities for loading or saving objects for a specific module. When an input or output button is pressed the following options become available:

Get from Pool shows all objects available in the Pool. Objects selected are converted for the active module if possible.

Put to Pool enters newly created objects into the Pool.

Show Displays objects, graphically if possible, otherwise simply as S-expressions;

Edit Edit the s-expression corresponding to the current object. Templates are provided to edit new objects.

In a typical interaction, the user first loads the module of interest, then clicks on the file button to load all the files available for that particular module. At this point he is able to load input (with appropriate conversions if possible) by clicking on the input button of his module. After loading input the module creates at least one output object. The user can then decide whether to save the output object in the Pool, thus making it available to every other active module.

4 The Future of Intermodule Communication

The Pool works reasonably well in simple cases, ensuring that every active data object available in memory can be shared, and allowing the exchange of data structures among different modules. When the user is sufficiently knowledgeable, the Pool is thus a useful tool for experimenting with different AI techniques.

The Pool mechanism for intermodule communication with some minor face-lifts seems to be more than adequate for naive to medium level users. However, experience suggests that expert users wishing to explore in-depth communication between modules need a far more sophisticated environment to solve their problems. In general, the data centric view to intermodule communication puts too many constraints on expert users. Interdisciplinary applications cannot be build only by sharing data among modules but some kind of process and data flow control should be built into the environment.

One example where the Pool approach to IC seems to fall short can be drawn from [4] and his description of learning classifier systems. A classifier system is a rule based system (1) capable of learning new rules in an arbitrary environment using a genetic algorithm (2) as a rule discovery system [4]. Most of the main parts to a classifier system are available individually in PAIL ((1) can be found in the rules module, (2) is available via the Genetic Alorithms module). Like many other applications however a learning classifier system routinely exchanges data between a rule system and the genetic algorithm. These cyclic constructs are not available through the Pool as is a Lisp entry point.

PAIL is object oriented and its key components (main function calls, data) are reusable and recombined to extend the functionality of the system. We envisage that the optimal solution for the creation of interdisciplinary applications would be a graphic engine capable of dealing with a variety of constructs such as input/output objects and parameters, cycle (loop) control, and Lisp-level routines.

We are also investigating the possibility of a metalanguage (macro language) whose expressions describe complex applications built out of the existing "primitive" modules. Here we envisage to apply the client/server metaphor while supplying the user with a set of primitives to send and receive messages from and to the modules core function calls.

5 Acknowledgements

The present state of the Portable AI Lab is a result of the contributions of many individuals. In particular we would wish to thank Dean Allemang, who was responsible for the original design of the Pool, and in addition Fatma Fekih, Nick Almassy, and Erik Vinkhuyzen. The work reported here has been in part funded by a research grant of the Swiss National Fund for Scientific Research.

References

[1] A.K.Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[2] D. Allemang and R. Aiken. Tailoring advanced instructional software for AI. In *IEA/AIE-92 Paderborn*, 1992.

[3] J. Doyle. A truth maintenance system. *Artificial Intelligence*, pages 231–272, 1979.

[4] J. H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell., editors, *Machine Learning*. Morgan Kaufmann, Los Altos, CA., 1986.

[5] Tom Mitchell, Richard Keller, and Smadar Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[6] Peter Norvig. *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1991.

[7] J. Ross Quinlan. *Machine Learning: An Artificial Intelligence Approach*, chapter Learning Efficient Classification Procedures and Their Application to Chess End Games. Tioga., Palo Alto, CA, 1983.

[8] Mildred Shaw and Brian Woodward. Validation in a knowledge support system: Construing and consistency with multiple experts. *Knowledge-based Systems*, 4:39–60, 1990.

[9] D. Waltz. Understanding line drawings with shadows. In *The psycology of computer vision*. Mc.Graw Hill, NY, 1975.