A Communication Language and the Design of a Diagnosis Agent – Towards a Framework for Mobile Diagnosis Agents*

Christian Piccardi and Franz Wotawa

Technische Universität Wien, Institut für Informationssysteme, Paniglgasse 16, A-1040 Wien, Austria, E-Mail {piccardi,wotawa}@dbai.tuwien.ac.at

Abstract. In this article we present a communication language and a design concept for a diagnosis agent. The communication language is divided in two parts. One part contains primitives necessary to control and query the agent. The other provides constructs for formulating the structure and behavior of systems to be diagnosed, and for observations determining the current state of the system. Additionally we give an overview of a WWW-based architecture for accessing an (diagnosis) agent's functionality. This approach is intended to make diagnosis tools accessible from todays WWW browsers.

1 Introduction

Model-based diagnosis (MBD) [Rei87,dKW87] has been successfully applied to several domains including the automotive industry for on- and off-board diagnosis of cars [MS96,PS97], for software debugging [FSW96,SW98], and tutoring systems [dKB98] among others. Only little effort has been spent so far on the design and implementation of diagnosis agents accessible using the WWW infrastructure. Currently most diagnosis systems use their own way to store, load, and process system models and observations, thus the prerequisite for the implementation of a WWW-based agent is a commonly agreed language for describing the behavior and structure of diagnosis systems, i.e., the system model, and the observations. A mobile diagnosis agent can be used in various domains ranging from academic institutions for educational purposes or in industry allowing easy maintenance and extendibility.

In this paper we describe the first part of the *MObile DIagnosiS (MODIS)* project. The objectives of MODIS are the design of a general framework for diagnosis, the exchange of diagnosis informations between agents, and the improvement of diagnosis algorithms to allow effective and efficient use in mobile computers such as personal digital assistants (PDAs) in case online communication is not possible. Our interest is to improve the algorithms to diagnose mid-size systems with up to 1,000 components using a PDA with limited computational power and memory.The objectives of the first part of MODIS are the design and implementation of a diagnosis agent for the WWW

* The work presented in this paper has been funded by grant H-00031/97 from the Hochschuljubiläumsstiftung der Stadt Wien. Christan Piccardi was supported by the Austrian Science Fund (FWF) under project grant P12344-INF.

using a communication language allowing to exchange diagnosis knowledge such as system descriptions and observations. The diagnosis agent itself has been implemented in Smalltalk. The client, i.e., the interface agent, has to be implemented using HTML and a Java applet. TCP/IP sockets are used as the underlying communication protocol.

During the last years several researchers have proposed different architectures of diagnosis agents. See for example [SDP96] or more recently [HLW98]. In [SDP96] an architecture of a diagnosis agent based on a non-monotonic inference systems is described. [HLW98] uses the communication language KQML ([FLM97]) and CORBA for communication. In contrast to previous approaches we do not base our system on one inference system. Instead, by using a communication module as interface between the diagnosis engine and requests from outside, the actual implementation of the diagnosis engine becomes less important, although the diagnosis engine has to implement all functions accessible from other agents. Additionally, we propose the use of standard WWW technology as the underlying communication architecture because of its availability on most of todays computer systems.

This paper is organized as follows. First, we give an introduction into model-based diagnosis followed by a section describing the agent's architecture and its design. Section 4 gives an overview of the communication language and the diagnosis functionality. A concise introduction into the used system description language for describing the behavior and the structure of a diagnosis system and the observations is given in section 5. A conclusion and future research directions are presented at the end of this paper.

2 Model-based diagnosis

In model-based diagnosis [Rei87,dKW87] a model of the *correct* behavior of a system or device is used to find a subset of system components that cause a detected misbehavior. This subset is termed a diagnosis. In contrast to other approaches, e.g., rule-based expert systems, no explicit effect-to-cause relationships have to be given to describe specific failure occurrences (although the model can be extended to describe typical incorrect behaviors). Very importantly, models can be composed of model fragments, i.e., behavioral descriptions of individual components such as an *AND* gate, merely by specifying the types of components and their interconnections. The model is thus easily maintainable. However, model-based diagnosis is only applicable if a model is available, which fortunately is the case in almost all technical domains.

A diagnosis is defined for a system (a device), consisting of a behavioral description SD and a set of components $COMP$, and a set of observations OBS. Formally (see [Rei87]), a set $\Delta \subseteq COMP$ is a diagnosis iff

$$SD \cup OBS \cup \{\neg ab(C) | C \in COMP \setminus \Delta\} \cup \{ab(C) | C \in \Delta\}$$

is consistent. The predicate $ab(C)$ is used to indicate that a component C is behaving abnormally. A correctly behaving component C is therefore described by $\neg ab(C)$.

We illustrate this diagnosis definition using the full adder example from figure 1. The behavior of the full adder is given by the behavior of its subcomponents (in this case, two inverters, two AND gates, one OR gate) and their interconnections. The correct behavior of an *AND* gate C can be formalized as follows:

$and(C) \Rightarrow (\neg ab(C) \rightarrow out(C) = andBehav(in_1(C), in_2(C)))$
where $andBehav(true, true) = true$, $andBehav(false, X) = false$, $andBehav(X, false) = false$.
Similar rules can be given for the *XOR* and the *OR* gate. The connection part of the system description is specified by:

$in_1(X2) = c_n.$ $in_2(X2) = out(X1).$ $out(X2) = q_{n+1}$
$in_1(X1) = a_n.$ $in_2(X1) = b_n.$
$in_1(A1) = b_n.$ $in_2(A1) = a_n.$ $out(A1) = in_2(O1).$
$in_1(A2) = a_n.$ $in_2(A2) = out(X1).$ $out(A1) = in_1(O1).$
$out(O1) = c_{n+1}.$

Now assume the input values $a_n = false, b_n = true, c_n = true$ have been used and the output values $q_{n+1} = true, c_{n+1} = true$ have been observed. Assuming the correct behavior of all components, i.e., $\neg ab(A1), \neg ab(A2), \neg ab(O1), \neg ab(X1), \neg ab(X2)$, the system description contradicts the observations. A model-based diagnosis algorithm such as the Hitting Set algorithm described in [Rei87], computes only one single diagnosis $ab(X2)$ (i.e., the failure of $X2$ alone would explain the observations). All other diagnoses consist of more than one faulty component.

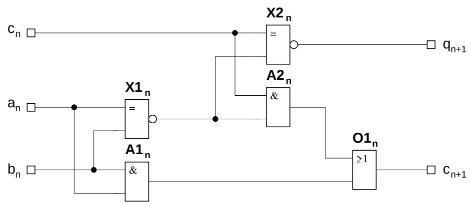


Fig. 1. A schematic of a full adder

In summary, the model-based diagnosis approach provides: (1) a precise diagnosis definition, (2) a clear separation between the knowledge about a system and the diagnosis algorithms, (3) a component oriented view that allows reuse of knowledge and easy maintenance, and (4) the use of multiple models. The only prerequisite for using model-based diagnosis is the existence of a component oriented system model. The diagnosis time depends on the number of system components. To give an upper bound for the diagnosis time we assume a model expressible in propositional logic where the size is a multiple of the number of components. The consistency of such a model can be checked in $o(|COMP|)$ time [Min88]. Searching only for single diagnoses results in at most $|COMP|$ consistency checks. Therefore the maximum time for computing all single diagnoses is of order $o(|COMP|^2)$. However, the expected runtime for diagnosis does not reach this bound in practice. The best diagnosis algorithms available today can diagnose systems with up to 10,000 components in less than one minute [FN97,SW97].

3 The diagnosis agent architecture

In order to provide easy access to an agent's capabilities the use of the WWW is recommended because of the provided functionality and the widespread use. We therefore introduce a global architecture for mobile diagnosis agents based on the WWW. The agent, that implements the diagnosis functionality, is executed on a dedicated server. The interface agent implemented as a Java applet embedded in a HTML page is executed on the client side. TCP/IP sockets are used for communication between them. Although the functionality of an applet is restricted for security purposes, socket communication between the applet and the server distributing the applet is allowed. Figure 2 gives an overview of the global architecture.

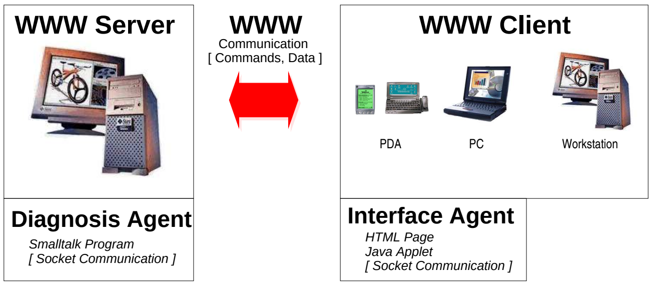


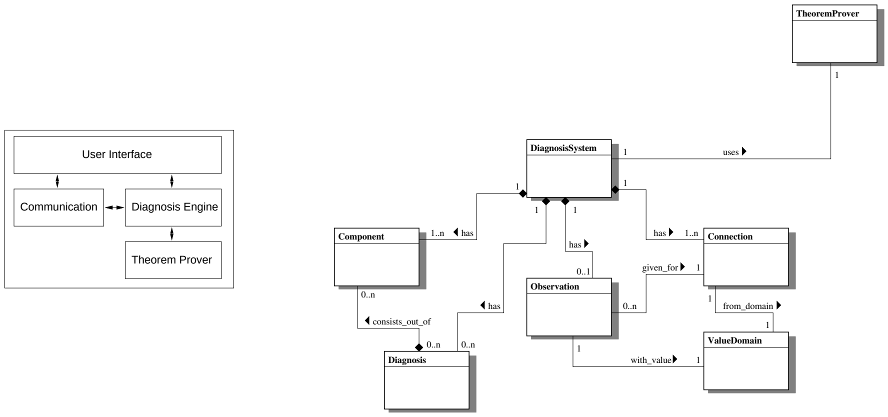
Fig. 2. The MODIS WWW architecture

As a consequence of the definition of model-based diagnosis, the diagnosis agent must include at least a diagnosis engine and a theorem prover. For interaction with other agents we additionally need a communication module and an user interface. Figure 3 (a) shows the agent's internal architecture.

The communication module controls the information exchange between agents, i.e., interface or other diagnosis agents. Its task is to parse the incoming messages and call the associated (diagnosis specific) functions together with their parameters. The result is coded using the communication language and sent back to the sender of the original message.

The diagnosis engine implements the whole diagnosis functionality such as diagnosis, probing (measurement selection), among others. Information about the behavior of the currently used diagnosis system and the corresponding observations are located within the theorem prover which is used for checking consistency while computing all diagnoses. Diagnosis parameters such as the maximum number of faulty components or the maximum number of diagnoses to be computed together with different diagnosis algorithms are part of the engine.

From the object-oriented point of view we have to deal with objects rather than modules. We illustrate the agent design by specifying some objects and their relations necessary for implementing the diagnosis engine. The most important entity for diagnosis is a system storing components, connections, observations, results and parame-



(a) (b)

Fig. 3. Diagnosis agent architecture (a) and diagnosis system design (b)

ters. Components of a specific type have an unique behavior. Values associated with connections must be from a given domain. Additionally, for diagnosis we need theorem provers which are specific for a given system. From this verbal description we can directly derive the objects DiagnosisSystem, Component, Connection, Observation, ValueDomain, Diagnosis, TheoremProver with the relationships given in figure 3 (b).

Because DiagnosisSystem has access to all diagnosis related informations we assume that it implements the whole diagnosis functionality, e.g., diagnosis or measurement selection. So a diagnosis engine can be implemented using DiagnosisSystems.

4 The communication and diagnosis requirements

One of the subtasks of MODIS has been the development of a language for the interchange of diagnosis informations between agents. In our view agents are programs implementing functionality accessible from the outside. The state of the agent is given by its internal data and is not necessarily accessible from outside. This approach is similar to [ESP98] and has the advantage of separating communication properties from their implementation. However, our approach is simpler but adequate for our purposes.

The communication language is divided in two parts. The first part contains those constructs necessary for communication functionality, i.e., controlling and querying the agent. The second part called *system description language (SDL)* is used to describe diagnosis systems and observations. Therefore the application domain of SDL is not restricted to agent communication. Instead it can also be used for storing diagnosis informations, e.g., building a model library. SDL is concisely described in section 5.

We assume that an agent $A_i = (i, F_i)$ has an unique identifier i and a set of functions F_i accessible by other agents. Communication is done by sending a function expression to an agent using the underlying communication protocol, i.e., sockets. It is assumed, that every time a function expression $f(a_1,\ldots,a_n)$, $f \in F_k$ of arity n is sent to an agent A_k, a reply from A_k is sent back. This reply contains the message $f(a_1,\ldots,a_n) = Result$ where $Result$ represents the resulting value of the function call. For example $loadSD(\ldots)$ will return $loadSD(\ldots) = done$ when the given system description has been successfully loaded. This hand-shake mechanism ensures proper synchronization between sender and receiver.

Using the above formalization a basic diagnosis agent D_i is defined by the tuple $(i, \{startDiagnosis, startProbing, loadSD, loadOBS, getSD, getOBS, getDiagnoses, getProbes, resetOBS, reply\})$. The functions have the following meaning:

Diagnosis At least diagnosis and measurement selection should be provided.
startDiagnosis() starts the diagnosis process using previously received informations about the system description and observations.
startProbing() runs the measurement selection routine. The system description, observations, and the derivable diagnoses must be available.

Access The system description, observations, diagnosis and measurement selection result must be accessible from the outside.
loadSD(SystemDescription) loads the system description into the diagnosis agent.
loadOBS(Observations) loads the observations into the diagnosis agent.
getSD() asks for the currently used system description.
getOBS() asks for the currently used observations.
getDiagnoses() asks for the last computed diagnoses.
getProbes() asks for the result of the measurement selection function.
resetOBS() removes the observations together with the diagnoses and measurement selection result.

A more general diagnosis agent can provide functions for setting and removing a component focus, or for computing informations based on system descriptions, e.g., a failure-mode and effects analysis (FMEA). In the next section the second part of the communication language is introduced that is used for exchanging diagnosis informations.

Since our diagnosis agent is intended to be used for educational purposes, i.e., teaching students model-based diagnosis, we add the functions for accessing predefined system descriptions and observations to the diagnosis agent:
allSDid() returns a list of names of system descriptions stored by the diagnosis agent.
getSD(SDIdentifier) returns a system description with the name *SDIdentifier*.
allOBS(SDIdentifier) returns a list of names of observations for the given system description named by *SDIdentifier*.
getOBS(SDIdentifier,OBSIdentifier) returns the observations named *OBSIdentifier* for a system *SDIdentifier*.

5 The system description language

The system description language (SDL) (see [Pic98] for the full syntax and semantics of SDL) is used for describing the properties of components and their interconnection.

It distinguishes between atomic and hierarchic components. In both cases it is necessary to specify the associated input, output or bidirectional ports which act as interfaces to other components. For atomic components the associated behavior modes are defined, whereas the behavior of hierarchical components is derived by their subcomponents and the connections between them.

<AtomicComponent> →
component <ComponentName> <PortDefinitions> [<VariableDefinitions>]
[<AxiomDefinitions>] <BehaviorDefinitions> **end component**
<HierarchicComponent> →
component <ComponentName> <PortDefinitions> <SubcomponentDefinitions>
<ConnectionDefinitions> **end component**

Each port of a component is associated with a type, determining the possible values that can be assigned to it. Predefined types are `integer, float, string, character` and `boolean`. Own types can easily be added by enumerating the corresponding elements or using the set-theoretic union of already existing types or arbitrary sets of elements.

Variables are used to allow for compact behavior definitions and are defined by associating a variable name with a set of elements. SDL only allows finite sets to be associated with variables.

Axioms are formulated in terms of predicate expressions, defining relations between objects for a given component. During the evaluation of a behavior mode, decisions based on components' axioms can be made. Axioms have a dynamic character and can be added or removed during mode evaluation thus having direct impact on a components behavior. Like the introduction of variables, the use of axioms allows for more efficient and compact behavior definitions.

The heart of each component is its behavioral model, stating how it behaves under certain circumstances. Each component in a system knows one or more different behavioral modes based on different assumptions. A behavior mode definition for an atomic component is given by the following syntax:

<BehaviorDefinition> →
[**default**] **behavior** <BehaviorName> [**prob**:<Probability>] [**cost**:<RepairCosts>]
[**action**:<RepairAction>] <RuleList> **end behavior**

For each behavior mode we know a probability stating how probable the occurence of this behavior is, an associated cost value telling how expensive the necessary repairs are and an action description stating which steps have to be taken in order to repair the component. The <RuleList>-production is used to formulate the causal dependencies and legal inference steps within the component.

The behavior of hierarchic components is derived from the behavior of their subcomponents and is based on the connections between them. For this reason a mechanism is needed to instantiate components within the current component definition, which is provided by the <SubcomponentsDefinition>-production. Names are associated with already defined components and can then be used to specify the connections between them.

<ConnectionDefinition> →
<ConnectionName> [**observable**:{**true** | **false**}]
[**cost**: <ObservationCost>] <ComponentPort> -> <ComponentPortList>.

Connections may or may not be observable and can have an associated cost value determining how expensive an observation would be. Each connection has its origin at the port of one component (<ComponentPort>) and can branch to many others (<ComponentPortList>).

As mentioned above the causal dependencies and legal inference steps within a component are formulated using rules. Each rule consists of a prerequisite and a consequence. Intuitively, if the prerequisite is true then the consequence can be infered. SDL uses two inference operators to allow for compact rule definitions.

<Rule> → [<Sentence>] =: <Sentence>. | <Sentence> :=: <Sentence>.

In the first case the right-hand side of the rule can be infered iff the left-hand side is true. In the second case inference is possible in both directions. If in the first case the left-hand side is empty the consequence can always be infered. Rules are defined in terms of sentences.

SDL distinguishes four types of sentences: *predicate sentences, conjunctive sentences, implication sentences* and *quantified sentences*.

<PredicateSentence> → <PredicateName>(<ArgumentList>)
<ConjunctiveSentence> → <Sentence>, <Sentence>
<ImplicationSentence> → <Sentence> -: <Sentence>
<QuantifiedSentence> → {**forall** | **exists**} <VariableList> : <Sentence>

<ArgumentList> is a possibly empty list of constants, ports, function expressions and variables.

When used in the prerequisite of a rule, we have to assign truth values to sentences, determining wheter the consequence can be infered. This is done as follows. viewing a predicate as a set of tuples, we define the truth value for a predicate sentence as *true* if and only if the tuple of objects determined by the arguments are an element of this set, otherwise as *false*. Given a conjunction the truth value is defined as true if and only if the truth value of every conjunct is true. For an implication s_1 -:s_2 the truth value is defined as follows: If s_1 is true then the truthvalue of the implication is true if and only if s_2 is true. If s_1 is false then the implication is true, regardless of the truthvalue of s_2 (ex falso quod libet). The truth-value for an universally quantified sentence is defined as true if and only if for all possible variable assignments the truth value of the sentence following the colon is true. Given an existentially quantified sentence the truth value is defined as true if and only if for at least one possible variable assignment the truth value of the sentence following the colon is true.

In order to provide a diagnosis tool with reference values for certain ports, we need a way to specify observations. We do this by specifying the observations and mapping them to a specific diagnosis component.

<ObservationDefinition> → **observation** <ObservationName> <PortAssignments>.
<PortAssignment> → <PortName> = <Value>.
<DiagnoseDefinition> → **diagnose**(<ComponentName>,<ObservationName>)

```
component XorGate
  input in1, in2 : bool.
  output out: bool.
  default behavior
    Neq(in1, in2) :=: Eq(out, true).
    Eq(in1, in2) :=: Eq(out, false).
  end behavior
end component
```

```
component AndGate
  input in1, in2 : bool.
  output out: bool.
  var Port: {in1, in2}.
  default behavior
    forall Port: Eq(Port, true) :=: Eq(out, true).
    exists Port: Eq(Port, false) =: Eq(out, false).
  end behavior
end component
```

```
component OrGate
  input in1, in2 : bool.
  output out: bool.
  var Port: {in1, in2}.
  default behavior
    exists Port: Eq(Port, true) =: Eq(out, true).
    forall Port: Eq(Port, false) :=: Eq(out, false).
  end behavior
end component
```

```
component FullAdder
  input a,b,c : bool.
  output q,z: bool.
  subcomponent
    and1, and2: AndGate.
    or1: OrGate.
    xor1, xor2: XorGate.
  connection
    s1: a -> xor1(in1), and1(in2).
    s2: b -> xor1(in2), and1(in1).
    s3: c -> and2(in1), xor2(in1).
    s4: xor1(out) -> and2(in2), xor2(in2).
    s5: and1(out) ->or1(in2).
    s6: and2(out) ->or1(in1).
    s7: xor2(out) -> q.
    s8: or1(out) -> z.
end component
```

Fig. 4. The SDL program implementing a full adder

Finally, we illustrate the capabilities of SDL using the full adder example depicted in figure 1. The full adder uses three types of boolean gates: a XorGate, an AndGate, and an OrGate. The SDL program describing the behavior of the three gates and the full adder consisting out of the gates is given in figure 4.

6 Conclusion

In this paper we have presented the architecture and design of a diagnosis agent accessible using a standard WWW browser. For communication purposes we have introduced a communication language consisting out of two parts. One for sending queries and requests to the agent and one for describing diagnosis systems and observations. Currently, the system description language and the diagnosis agent and parts of the WWW interface have been implemented.

In summary the paper provides: (1) a general architecture for diagnosis agents, (2) a description of a WWW interface for a diagnosis agent, and (3) a communication language for controlling and querying an agent, and exchanging and storing diagnosis informations. Future research in this domain include the improvement of diagnosis algorithms, their implementation within the introduced framework, and the design of a framework for a graphical user interface especially for diagnosis.

References

[dKB98] Kees de Koning and Bert Bredeweg. Using GDE in Educational Systems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 279–283, Brighton, UK, August 1998.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[ESP98] Thomas Eiter, V.S. Subrahmanian, and George Pick. Heterogeneous active agents. Technical report, Institut für Informatik, Justus-Liebig-Universität Gießen, Germany, IFIG Research Report 9802, March 1998.

[FLM97] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an Agent Communication Language. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 291–317. AAAI Press / The MIT Press, 1997.

[FN97] Peter Fröhlich and Wolfgang Nejdl. A Static Model-Based Engine for Model-Based Reasoning. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, Nagoya, Japan, August 1997.

[FSW96] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Budapest, August 1996.

[HLW98] Florentin Heck, Thomas Laengle, and Heinz Woern. A Multi-Agent Based Monitoring and Diagnosis System for Industrial Components. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, pages 63–69, Cape Cod, Massachusetts, USA, May 1998.

[Min88] Michel Minoux. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters*, 29:1–12, 1988.

[MS96] A. Malik and P. Struss. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, 1996.

[Pic98] Christian Piccardi. AD^2L An Abstract Modelling Language for Diagnosis Systems. Master's thesis, TU Vienna, 1998.

[PS97] Chris Price and Neal Snooke. Challenges for qualitative electrical reasoning in circuit simulation. In *Proceedings of the 11th International Workshop on Qualitative Reasoning*, 1997.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[SDP96] M. Schroeder, C.V. Damasio, and L.M. Pereira. REVISE Report: An Architecture for a Diagnosis Agent. In *ECAI-96 Workshop on Integrating Non-Monotonicity into Automated Reasoning Systems*, Budapest, Hungaria, August 1996.

[SW97] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, Nagoya, Japan, 1997.

[SW98] Markus Stumptner and Franz Wotawa. VHDLDIAG+:Value-level Diagnosis of VHDL Programs. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, Cape Cod, May 1998.