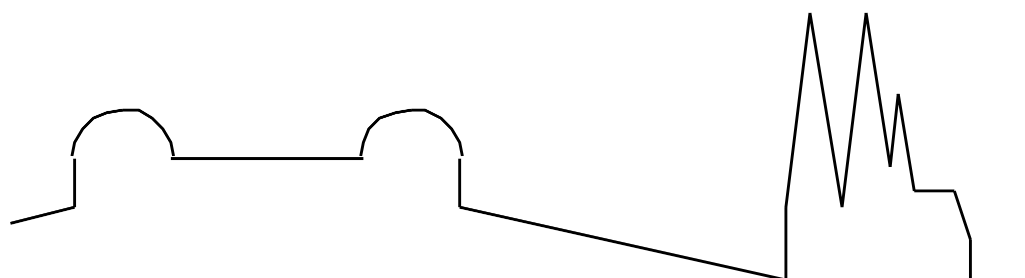
UPPSALA THESES IN COMPUTING SCIENCE
No. 27/97

Integrating Heuristic and Model-Based Diagnosis

Kent Andersson



Thesis for the Degree of
Licentiate of Philosophy

UPMAIL
Computing Science Department
Uppsala University
Box 311
S–751 05 UPPSALA
Sweden

ISSN 0283–359X

UPPSALA THESES IN COMPUTING SCIENCE
No. 27/97

Integrating Heuristic and Model-Based Diagnosis

Kent Andersson
UPMAIL, Computing Science Department
Uppsala University
P.O. Box 311, S–751 05 Uppsala
Sweden

ABSTRACT

We discuss the prospects for combining heuristic and model-based diagnosis in an integrated diagnosis system. By introducing a metalogic representation structure with a metatheory and an object theory, an integration is achieved that separates the model-based and the heuristic knowledge while allowing it to be used together. The object theory of the system consists of the model-based knowledge whereas the metatheory consists of (i) heuristic knowledge, (ii) knowledge compiled from principles of the domain but not represented in the object theory for reasons of complexity, and (iii) a strategy for computing diagnoses. A graphical interface for the diagnosis system is presented, supporting the user in giving observations by taking advantage of the model-based knowledge. Finally, the possibilities for building diagnosis shells integrating model-based and heuristic diagnosis for classes of domains are discussed.

Thesis for the Degree of
Licentiate of Philosophy
ISSN 0283–359X

This thesis is dedicated to my parents and my family

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge my debt to the people involved, directly or indirectly, in the present investigation. It has been carried out at the Computing Science Department of Uppsala University. First, I wish to express my sincere gratitude to its head Professor Åke Hansson for his useful advice, many stimulating discussions, never failing interest and constant encouragement throughout this work.

I am greatly indebted to Professor Keith Clark for valuable comments and suggestions on earlier drafts of this thesis. I am most grateful to the following persons. To Dr. Torkel Hjerpe for many interesting discussions, sound advice, pleasant cooperation on various projects and helpful linguistic counsel. To the members of the department for making it a pleasant, stimulating and interesting place to work.

Finally, I am deeply grateful to my family and parents for their loving support and encouragement.

Uppsala Kent Andersson
April 1997

Contents

1 INTRODUCTION 1
1.1 Purpose 2
1.2 Outline of the Thesis 2
1.3 Typographical Conventions 2

2 DIAGNOSIS SYSTEMS 3
2.1 A Classification of Diagnosis Approaches 3
2.2 GDE 5
2.3 A Theory of Diagnosis for Incomplete Causal Models 7
2.4 Sherlock 10
2.5 Properties of Diagnosis Problem Solving 11
2.6 Promises and Problems of Model-Based Diagnosis 12

3 DIAGNOSIS IN A METALOGIC SYSTEM 15
3.1 Logic Knowledge Representation 15
3.2 Metalogic Knowledge Representation 16
3.3 A Metalogic Diagnosis System 16

4 FORMALIZATION OF A DIAGNOSIS DOMAIN 19
4.1 An Example Domain 19
4.2 Overview of the Object Theory 20
4.3 Structure 20
4.3.1 Objects 21
4.3.2 Configuration 23
4.4 Function 24
4.4.1 Output 24
4.4.2 Input 25
4.4.3 Transformation 27
4.5 Derivation from the Theory 30

5 A DIAGNOSIS THEORY 31
5.1 Observations 32
5.2 Hypotheses 33
5.2.1 Abnormal Observations 33
5.2.2 The Signal Graph 36
5.2.3 The Generation of Hypotheses 38
5.3 Object Level Refutation 40
5.4 Metalevel Refutation 43
5.4.1 Two Kinds of Compiled Knowledge 43
5.4.2 A Representation of Complex Compiled Knowledge 45
5.4.3 A Representation of Heuristic Compiled Knowledge 49
5.5 A Representation of a Diagnosis Strategy 50
5.5.1 A Definition of Diagnosis 51
5.5.2 A Diagnosis Strategy 51
5.5.3 A Representation of the Diagnosis Strategy as a Logic Program 54
5.5.4 Diagnosis of Multiple Dependent Faults 55
5.6 Summary 56

6 AN EXAMPLE DIAGNOSIS CASE 57
6.1 Introduction 57
6.2 The Computation of a Diagnosis 57

7 INTERACTING WITH THE USER 63
7.1 Properties of a User Interface 63
7.2 Design of a Graphical User Interface for a Diagnosis System 65
7.3 A Representation of Knowledge for a Graphical User Interface 67

8 FUTURE WORK 71
8.1 Generalization of the Diagnosis System 71
8.2 Computer Support for the Construction of Diagnosis Systems 74

9 CONCLUSIONS 75

REFERENCES 77

Chapter 1

Introduction

Diagnosis is a complex problem solving task with different kinds of knowledge that interact. In diagnosis we find several kinds of knowledge—principled domain knowledge, heuristic knowledge, control knowledge, diagnosis strategy knowledge and knowledge about the user's answers. The early diagnosis systems, such as MYCIN in [17], used heuristic knowledge for making diagnoses. The heuristics mainly consisted of empirical associations from symptoms to causes, where the use of domain specific knowledge was the key to the success of these systems, but also one of their main weaknesses. The representation of the heuristics included both domain and control knowledge—a mix that became unmanageable when the knowledge base became large and necessary to revise [5]. Another problem with the heuristic approach is that it can be difficult to elicit the empirical knowledge that is needed for the rules [24, 29, 27]. In response to the knowledge elicitation problem, a new type of diagnosis system evolved that did not include empirical knowledge about symptoms and diagnoses, but relied on principled knowledge of the domain. A diagnosis system of this type is called model-based, because it has a model of the diagnosis object, which describes the construction and function of the object. A model-based diagnosis system applies a general diagnosis algorithm to its model to find a diagnosis. Originally, this type of system did not use heuristic knowledge about the domain to guide the diagnosis, because the knowledge in the model was principled and therefore not to be mixed with heuristics. As a consequence, the general diagnosis algorithm did not take advantage of the short-cuts that heuristic knowledge can provide. A diagnosis system that integrates model-based and heuristic diagnosis knowledge could take advantage of both kinds of knowledge.

When we try to represent different kinds of problem solving knowledge in the same program a problem arises. It manifests itself as difficulties to make the various kinds of knowledge distinct as well as work together seamlessly in the program. If we use the same representation for domain and control knowledge and mix them to make them work together, we get a computationally efficient but flat representation that does not preserve the logical structure of the knowledge. An example is a rule based representation of domain knowledge where control knowledge is represented as conditions of the rules. Such a mixture results in rules that are not only about factual domain knowledge, but also about when

the domain rules should be tested. When we want to change the rules, either add rules or change facts in them, it will be difficult to do so because factual knowledge is mixed with control knowledge which can be difficult to distinguish. Thus, a problem with flat representations is a lack of transparency, resulting in programs that are difficult to change. Furthermore, in a flat representation we cannot represent a structure where one program discusses another program, leading to difficulties to represent some kinds of knowledge that discusses other knowledge, such as control and domain knowledge, or strategy and domain knowledge. In contrast, if we separate the representation of different kinds of knowledge, we can build a representation that preserves the logical structure and transparency of the knowledge. However, the problem of making the different kinds of knowledge work together still remains.

1.1 PURPOSE

This thesis studies the problem of representing knowledge for a diagnosis system. The purpose is to investigate the prospects for representing problem solving knowledge such that its logical structure is preserved, it is represented in a modular fashion allowing parts to be replaced or modified without affecting other, the representation is transparent enough for the knowledge engineer to distinguish different kinds and the representation allows them to be used together. We propose a metalogic representation structure in which we separate the different kinds of knowledge in order to preserve its logical structure and to make it possible to revise and edit.

Metalogic is a good candidate for a structured representation of diverse kinds of knowledge because it provides a division of logical levels of knowledge into logic levels of representation. The methodology used for the representation is therefore based on metalogic. Sentences in an object theory represent the domain knowledge for a device to be diagnosed, while sentences in a metatheory represent a diagnosis strategy, heuristic knowledge and control knowledge. The communication of computations of the object theory is performed by a predicate *demo* that relates the object theory and names for sentences provable in the object theory.

1.2 OUTLINE OF THE THESIS

In Chapter 2 we describe different approaches to diagnosis and discuss properties of the diagnosis problem. Chapter 3 provides an introduction to the representation methods used in the thesis. Chapter 4 discusses a representation of model-based knowledge for an example domain. Chapter 5 investigates the prospects of giving a representation of a diagnosis strategy and heuristic knowledge separated from the domain knowledge, and discusses computation with the representation. In Chapter 6 the computation of a diagnosis case is presented, and the design of a user interface for a diagnosis system is discussed in Chapter 7. Chapter 8 discusses the future development potential of the representation approach. Chapter 9, finally, concludes the thesis.

1.3 TYPOGRAPHICAL CONVENTIONS

Object theory formulas are written in typewriter font: `component(X)`, whereas formulas of the metatheory are given in italics: *normal(O, T)*.

Chapter 2

Diagnosis Systems

To introduce the reader to the area of diagnosis we classify the major expert system approaches and describe three examples of model-based diagnosis systems in some detail. We also discuss common properties of the approaches in order to identify some central parts of the diagnosis process. Finally, we discuss the promises and problems of model-based diagnosis.

2.1 A CLASSIFICATION OF DIAGNOSIS APPROACHES

The various approaches to diagnosis can be classified into two types: (1) First Generation Expert Systems and (2) Second Generation Expert Systems, where the main difference concerns what kind of knowledge is represented in the knowledge base. Each type can be further divided into finer categories, as will be developed below. This classification is not developed in full detail, but will serve to put the thesis into context.

First Generation Expert Systems

Domain specific systems

A domain specific expert system represents expert problem-solving knowledge for a particular domain, for example as rules or frames. To exemplify, we discuss rule based systems. The knowledge base consists mostly of associative rules relating symptoms to disorders or faults—heuristic knowledge. A simple inference mechanism is used to infer disorders from the rules. To accomplish the desired problem solving behaviour control and strategy knowledge is mixed with factual domain knowledge in these rules. The classical diagnosis system MYCIN in [17] exemplifies the first-generation approach to diagnosis. It diagnoses blood disorders using associative rules that relate symptoms to disorders. Consider the following rule (from [44]):

IF

1. A complete blood count is available
2. The white blood count is less than 2500

THEN

The following bacteria might be causing the infection

E. Coli (.75)

Psuedomonas-aeruginosa (.5)

Klebesiella-pneumoniae (.5)

This rule can be described as surface knowledge—it contains just enough information to have the effect of the required inference, but no underlying (deep) domain knowledge. This kind of knowledge can be called compiled since it has been transformed from general deep knowledge to specific. For example, the causal relation between a compromised host (a generalization of the concept 'white blood count less than 2500') and gram-negative infections (E.Coli is an instance of the class of gram-negative infections) is not represented explicitly. This knowledge is only implicitly present in the rule as a compilation of the deep domain knowledge underlying the rule. As a result of representing such specific rules the system has a very effective diagnostic capacity for cases that are included in the knowledge base, but cases not well covered by the associative rules cannot be handled. Such a system is often called "brittle" since it breaks easily when facing a problem outside the range it was designed for [44]. The explanation of the system's reasoning may be difficult to follow since its is usually based on the rules that represent surface knowledge, not on the deep knowledge behind the rules. Furthermore, the system may be difficult to maintain since the deep domain knowledge, which constitute the principles behind the rules, is not explicitly expressed in the knowledge base. Finally, these systems can be difficult to revise and extend because control knowledge, such as screening clauses (a clause that prevents the simple inference mechanism from asking unnecessary questions), is mixed with domain knowledge, quickly making the knowledge base complex and confusing [5].

Expert System Shells

Expert System Shells is a development of the domain specific systems. The idea is to remove the domain specific rules from an existing system and fill the empty knowledge base with knowledge from another domain. The domain knowledge is still represented as associative rules but comes from a different domain. The inference procedure is the same as in the original system. An example expert system shell is EMYCIN in [49], which was derived from the MYCIN system.

Second Generation Expert Systems

Explicit Strategy Systems

Systems with an explicit strategy for diagnosis replace the simple inference mechanism of the first-generation systems with an explicit strategy of some sophistication. One of the first examples of this type of system is NEOMYCIN [8]. This is an expansion and restructuring of MYCIN for the purpose of teaching. The weakly focused exhaustive search of the rules is replaced by an explicit diagnostic strategy. This strategy is represented separately from the domain knowledge which still has a rule based representation in the knowledge base.

Similar developments in this category include inference structures [7], problem-solving methods [37] and generic tasks [6].

Model-Based Systems

In model-based systems (sometimes called deep expert systems) the knowledge base contains a model of the domain and an inference calculus is applied to the model. The model can be a causal, functional, structural or other type of model [44], and represents only knowledge about correct behaviour, or only about abnormal behaviour, or a combination of both. A diagnosis is computed by deriving consequences of the model using the inference calculus and comparing the actual measurements in the diagnosis case.

We distinguish a group of model-based systems for diagnosis (MBD) that use logic for representation and reasoning. This group can be subdivided into three groups:

Consistency-Based MBD

In consistency-based MBD only knowledge about the structure and function of correct components is represented in the knowledge base. Examples of this approach include Genesereth [22] Reiter [41] and GDE in [19].

Abductive MBD

Systems with abductive MBD use only knowledge about faults and symptoms in the model—only knowledge about faulty behaviour. Examples of this approach include Poole et al. [39, 40], Cox et al. [15] and Console et al. [11].

Integrated MBD

Finally, integrated MBD is a combination of the abductive and the consistency-based approach that uses models of both correct and faulty behaviour. Examples in this category are: Sherlock in [20] and Console et al. [13].

In the following we will give an example of each of the three approaches to model-based diagnosis identified above, in some detail. See also [16, 26] for surveys of model-based diagnosis.

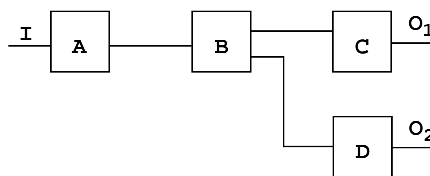
2.2 GDE

One of the best known systems for model-based diagnosis is the General Diagnostic Engine (GDE) developed by de Kleer and Williams [19]. It is an example of the consistency-based approach. The system is an improvement on earlier model-based systems, e.g. DART in [22] and SOPHIE [3] which diagnosed single faults in digital and analogue circuits respectively. GDE employs techniques which allow it to find multiple faults in circuits.

In outline, GDE works as follows. A circuit is represented as (1) a description of the physical structure of the circuit, and (2) a model for each component that makes up the circuit, describing the function of the component, e.g. as a constraint. No information on possible fault behaviour is represented. A general diagnostic procedure uses this representation of a correct device to produce a diagnosis for a set of measurements. The procedure is based on propagating values through the models for the device to predict output values, detecting predictions inconsistent with measurements and

finding sets of faulty components that would account for the inconsistencies. Probabilities for component failure are used to guide what measurements to take to discriminate between candidates. An assumption-based truth maintenance system (ATMS) [18] together with control strategies for the inference procedure is used to minimize the number of inferences that are necessary to find multiple faults in a device. For example, inferences for predictions of device behaviour are recorded in the ATMS, to avoid doing the same inference more than once. Furthermore, by using minimal environments to generate minimal candidates, as explained below, GDE reduces the computational complexity inherent in diagnosing multiple faults. In practice it seems that the number of multiple faults is usually quite limited (1–3 components), which limits the computational complexity of the diagnosis process.

As an example of GDE's diagnosis process, consider a very simple circuit of four components, A, B, C and D, as depicted below. GDE requires a model for each of the components, describing the



relationship between input and output values of the component. Let us assume that we have an inconsistency for the output O_1 of component *C*, where the predicted value has been propagated through components *A*, *B* and *C*. This would generate a conflict of three components ⟨*A,B,C*⟩ from which GDE would first generate three minimal candidates: [*A*], [*B*] and [*C*], since each singleton would explain the inconsistency. Assume next that we measure a new inconsistency at the output O_2 of component *D*, which generates a new conflict ⟨*A,B,D*⟩. This conflict would eliminate the minimal candidate [*C*] since it can not explain the new conflict. However, its immediate supersets in the candidate space (see below) would be examined to investigate if they could explain the conflicts. The supersets of [*C*] are: [*A,C*], [*B,C*] and [*C,D*], where the first two are not minimal since we have the minimal candidates [*A*] and [*B*]. They respectively subsume candidates [*A,C*] and [*B,C*], which therefore are eliminated. Remaining minimal candidates would be [*A*], [*B*] and [*C,D*]. So, the final result of the diagnosis, barring further measurements, would be that either *A*, *B*, or both *C* and *D* are the faulty components.

In more detail, GDE starts by propagating the current input values to the device through the device models to predict output values of the device. Each predicted value has a *supporting environment* which consists of the (correctness assumptions of the) components that were used to propagate the value. When a measurement is added to the database that is inconsistent with a value predicted by GDE, a *conflict* is generated by the system. The conflict consists of the supporting environment for the inconsistent prediction, now indicating that at least one of the components in the environment is faulty. By propagating values through the smallest environments first, the conflicts produced will be *minimal conflicts*, meaning that no subset of the conflict will also be a conflict. Using the set of minimal conflicts GDE generates *minimal candidates* that account for the conflicts, i.e. a minimal candidate is the minimal set of failed components that can explain a conflict. The candidates are constructed by adding components from conflicts to previously generated candidates, starting with the empty candidate. The set of possible candidates—*the candidate space*—is viewed as a lattice where the top element is the candidate consisting of all components in the device, and the smaller elements are the possible subsets of the top element. The lattice is used to construct new minimal candidates when new measurements are added to the system during diagnosis, by examining the supersets of the old candidates. The diagnostic process of GDE is repeated iteratively through a cycle of prediction, conflict detection, candidate

generation and measurement suggestion until the diagnostic objective has been achieved, for example that only one diagnostic candidate remains.

When more than one candidate remains, new measurements must be taken to distinguish between candidates. GDE forms its suggestions using a one-step lookahead strategy based on probabilities. New measurements are proposed by examining the conditional probability for each minimal candidate (given the hypothesized measurements) to determine a measurement that will minimize the entropy of the candidate probabilities, i.e. that will best distinguish among the candidates.

Entropy, H, is defined as:

$$H = -\sum p_i \log p_i \ ,$$

where p_i is the probability that candidate C_i is the actual candidate given a proposed measurement. Before any new measurements are taken, each candidate is equally likely—the entropy is maximal. Entropy is minimal when one candidate is much more likely than others. The best measurement is the one that minimizes the expected entropy of candidate probabilities, i.e. the measurement that is expected to mostly increase the probability of some candidates and decrease it for others.

Expected entropy, $H_e(x_i)$ of the candidate probabilities after measuring x_i is defined as:

$$H_e(x_i) = \sum_{k=1}^{m} p(x_i = v_{ik}) H(x_i = v_{ik}) \ .$$

For each of the m possible values for x_i the value of $H(x_i = v_{ik})$, i.e. the entropy resulting if x_i is measured to be v_{ik}, is computed by determining the new candidate probabilities, p'_l from the current candidate probabilities. p'_l for a candidate l is given by:

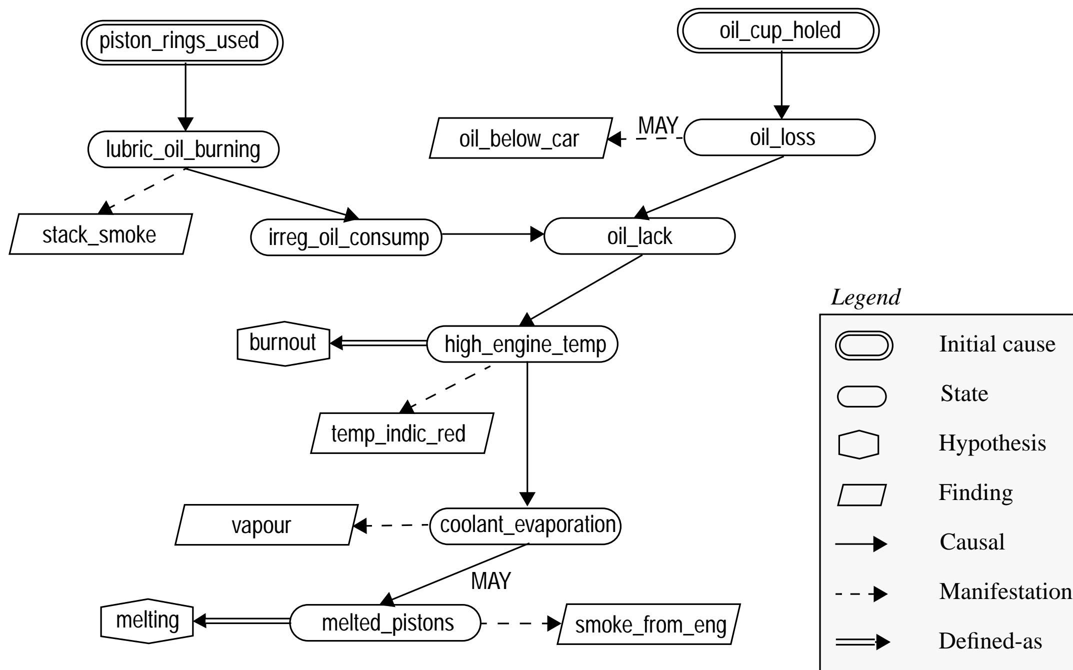
$$p'_l = \begin{cases} \dfrac{p_l}{p(x_i = v_{ik})}, & l \in S_{ik} \ , \\ \dfrac{p_l / m}{p(x_i = v_{ik})}, & l \in U_i \ , \end{cases}$$

where S_{ik} are the candidates where x_i, must be v_{ik}, and U are the candidates that do not predict a value for x_i. If all candidates predict a value for x_i then $p(x_i = v_{ik})$ is the combined probability for all candidates predicting $x_i = v_{ik}$. If some candidate does not predict a value for x_i then $p(x_i = v_{ik})$ is approximated by assuming that all m possible values for x_i are equally likely. The *initial probability* p_l for a candidate l is the product of the probabilities of failure for each component that is part of the candidate. This is given beforehand, for example by the manufacturer. The above formula for new candidate probabilities has been obtained via Bayes' rule for conditional probability.

2.3 A THEORY OF DIAGNOSIS FOR INCOMPLETE CAUSAL MODELS

An example of the abductive approach to model-based diagnosis is Console et al.'s theory of diagnosis for incomplete causal models [11]. In this approach diagnosis is viewed as an abductive process starting from a causal model that models the *faulty* behaviour of a system. We have only a fault model and no information of correct behaviour is used in the diagnosis. The general idea of abduction in diagnosis is to model the domain as implications from faults to symptoms, and then try to find those faults that would imply the observed symptoms.

The causal model is represented as a network of nodes and arcs. Cause-effect relationships are represented in the network as arcs from initial cause nodes to internal state nodes. A cause-effect relationship between an internal state and its observable manifestation is represented as a broken arc from a state node to an observable finding node. A relationship can also have a MAY-condition indicating that the model of the relationship is incomplete, i.e. the relationship is not fully specified. Diagnostic hypotheses are concepts defined in terms of internal states and represented as double arcs from state nodes to hypothesis nodes. An example adapted from [11] illustrates the representation.



In this network we can, for example, see that the finding *stack_smoke* is an observable manifestation of the internal state *lubric_oil_burning*, and that the initial cause of this is *piston_rings_used*. When more than one arc enters a node, e.g. *oil_lack*, this represents a disjunction. Nodes can also have attributes that further characterize the node, e.g. *lubric_oil_burning* has an attribute *quantity*, and *stack_smoke* has an attribute *color*. A function that represents how an attribute's value depends on the causing node's attribute, is associated with each causal and manifestation arc.

The causal model is formalized in logic as implications from initial causes to findings, where the nodes are represented as atomic formulas. Attributes are represented as arguments of the atomic formulas and functions for the attributes are defined. A MAY-condition is formalized as a conjunct in the antecedent of an implication. Diagnostic hypotheses are formalized in terms of states, as we will see below in an example.

As an example of the formalization of a causal model, consider the top left part of the network above. This would be formalized in logic as follows.

$$piston_rings_used \rightarrow lubric_oil_burning(x)$$
$$lubric_oil_burning(x) \rightarrow stack_smoke(f_1(x))$$
$$f_1(low) = grey$$
$$f_1(high) = black$$

The goal of the diagnosis is to find a causal explanation for the observed findings, by identifying a set of assumptions which, given the causal model, can explain the observed findings. An assumption is either an initial cause or a MAY-condition of an arc in the network. By adding the set Ψ of ground manifestation atoms of a particular diagnosis case to the logic representation of the causal network, we get a *diagnostic problem*

$$P = \langle NET, HYP, \Psi \rangle,$$

where *NET* is the set of logical formulas representing the network, *HYP* is the set of pairs $\langle H, def(H) \rangle$ such that *H* is a hypothesis node in the network and *def(H)* denotes the existential closure of a conjunction of the states leading to *H* in the network. Diagnosis is performed by constructing a world, *W* for *P*—a tentative causal explanation of the observed findings.

Given a diagnostic problem *P*, *W* is a *world* for *P* if and only if

$W = NET \cup \{\sigma_1, ..., \sigma_k\}$, where each σ_i is a ground initial cause atom in *NET*

or

$W = W' \cup \{\alpha\}$, where
- W' is a world for *P*,
- $X \wedge \alpha \rightarrow Y$ is an instance of a formula in *NET*,
- α is a MAY-condition symbol,
- $W' \vdash X$.

The central concept in the diagnosis process is that of a final world. The notion of covering is used to find a final world that explains all the observed findings in the diagnosis case. This means that all observed findings should be derivable from the final world. A further requirement is that no unobserved findings be derivable from the world.

Given a diagnostic problem *P*, *W* is a *final world* for *P* if and only if

$W \vdash m$ for all ground manifestation atoms $m \in \Psi$ (covering), **(2-1)**

it is not the case that $W \vdash m$ for some ground manifestation atom $m \notin \Psi$

Finally, the solution to a diagnostic problem *P* is viewed as the set of hypotheses that a causal explanation in a final world leads to. Given a final world *W*, a *solution* Θ to *P* is

$$diagnosis(W) = \{H \mid \langle H, def(H) \rangle \in HYP, W \vdash def(H)\},$$

where *def(H)* is the existential closure of a conjunction of the states leading to *H* in the network.

Consider the following diagnostic problem given the causal network illustrated above.

$$P_1 = \langle NET_1, HYP_1, \Psi_1 \rangle,$$

where NET_1 is the set of logical formulas representing the network,

$HYP_1 = \{\langle burnout, high_engine_temp \rangle, \langle melting, melted_pistons \rangle\}$,

Ψ_1 = {*temp_indic_red*, *vapour*}.

The only final world for P_1 is W_1 = {*oil_cup_holed*} since it is the only initial cause that explains the observed findings *temp_indic_red* and *vapour*, as well as it does not allow us to derive the unobserved finding *stack_smoke*. The solution corresponding to W_1 is Θ_1 = {*burnout*}, since it is the only hypothesis connected to the state *high_engine_temp* that is derivable from W_1. The hypothesis *melting* (and its defining state *melted_pistons*) is not in the solution. The reason is that the inclusion of *melted_pistons* (and its MAY-condition) into the final world would allow us to derive the unobserved finding *smoke_from_eng*.

In [13] the diagnosis approaches of abductive reasoning and consistency-based reasoning are integrated. This is done by incorporating a model of correct behaviour of the system to be diagnosed into a causal network model of faulty behaviour. The fundamental idea is to demand that abnormality observations must be covered (in the sense of formula (2-1)) and normality observations are only checked for consistency in a diagnostic explanation. The authors claim to produce a uniform framework in which many definitions of diagnosis can be captured. See also [39] for a comparison of an abductive and a consistency-based approach in the Theorist [40] framework.

Another example of model-based diagnosis with causal models is [38] which uses several *scenario models*, generated from a set of causal relationships for the domain, to diagnose performance problems in a computer system. Each scenario model is a causal model that identifies a diagnosis to which constraints are associated. A constraint can be assumptions and conditions needed to be satisfied when applying a scenario model. The strategy is to test which scenario model that best explains the observations.

2.4 SHERLOCK

Sherlock in [20] is an extension of the General Diagnostic Engine (GDE) in [19] with the inclusion of information about faults in the model. It is thus an example of the integrated approach to model-based diagnosis. Diagnosis is viewed as the task of identifying what exact state a component is in, not only to identify it as correct or faulty.

Each component is axiomatized by its behaviour, in *behavioural modes*, where a mode can be a normal state or a faulty state. Given a library of such axioms for components and a device structure, Sherlock constructs a set of axioms—the system description—describing the total behaviour of the device. The aim of the diagnosis is to find an assignment of a behavioural mode to each component (a *candidate*), such that for every observation, the union of the assignments, the system description and the observation is logically consistent. We illustrate with an example from [20]: The axioms below describe the behavioural modes for an inverter. *G* denotes “good”, *S1* denotes “output stuck-at-1”, *S0* denotes “output stuck-at-0”. In addition, every component has an “unknown mode” *U* that represents all modes of failure where the behaviour is not known.

Inverter Axioms

$INVERTER(x) \rightarrow [$

$\quad [G(x) \rightarrow [IN(x) = 0 \equiv OUT(x) = 1] \wedge$

$\quad [S1(x) \rightarrow [OUT(x) = 1] \wedge$

$\quad [S0(x) \rightarrow [OUT(x) = 0]\ ]$

In Fig. 2-1 below we see a device consisting of three inverters *A*, *B* and *C*, with input *I*, output *O* and possible measurement points *X* and *Y*. A diagnosis for this circuit, with the observation {*I=1, X=0, O=1*} would be [*G(A), G(B), S1(C)*] meaning that inverter *C* is faulty (with output stuck at 1). In computing this diagnosis Sherlock takes the inverter axioms and a description of the device structure to construct a system description from which a diagnosis is computed.

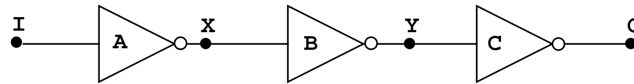


Fig. 2-1: *A three-inverter circuit.*

Sherlock computes its diagnoses in an assumption-based truth maintenance system (ATMS) [18] by constructing a set of *conflicts*, where a conflict is an assignment of behavioural modes to the components that is inconsistent with the system description and some observation. That is to say, the conclusions that can be drawn from the assignments are inconsistent with the observations. For example, given the system description for the three-inverter circuit in Fig. 2-1, the assignment [*G(A), G(B), G(C)*] and the observation {*I=1, O=1*} are inconsistent since we can derive *O=0* from the system description and *O=1* is in the observation. A *diagnosis* is an assignment that does not contain any minimal conflict, i.e. the minimal number of assignments that is still a conflict.

Like GDE, Sherlock uses probability measures in combination with Bayes rule to control the combinatorial explosion of the number of diagnoses that it otherwise would need to reason with. Each behavioural mode for each component is assigned a probability, for instance *p(S1(A))=0.008*, *p(G(A))=0.99*, *p(S0(A))=0.001* and *p(U(A))=0.001*. Only the most probable diagnoses are considered in the reasoning, using focusing tactics.

Sherlock is currently in practical use for diagnosing photo copiers [21].

Other researchers have also studied the integration of fault models in consistency-based diagnosis. One example is an extension of the GDE, GDE+ in [46] which uses fault models to generate only diagnosis candidates that are consistent with some fault model. In contrast to Sherlock, GDE+ does not use probabilistic information to guide the search. Instead control strategies are employed. Other examples of the use of fault models are Holtzblatt's GMODS system [32] and Hamscher's XDE [25] which deal with complex sequential digital circuits that have states.

2.5 PROPERTIES OF DIAGNOSIS PROBLEM SOLVING

In order to identify some central parts of the diagnosis process, we now discuss the common properties of the diagnosis approaches presented above.

Observations In all of the approaches presented above observations play a central role. The observations drive the diagnostic process forward. In GDE they are referred to as measurements, and in Console et al.'s system they are called findings.

Hypotheses Based on one or several observations all of the systems generate a set of diagnostic hypotheses. In GDE they are called candidates and in Console et al.'s system they are called worlds.

Refutation of hypotheses The diagnosis process of the systems presented is based on ruling out the diagnostic hypotheses by comparing user observations with model-based predictions for each hypothesis.

Ordering of hypotheses to refute In all systems focusing strategies are discussed. In GDE probabilistic information is used to guide the selection of candidates to refute.

User interaction Although not explicitly discussed in any of the systems, user interaction is always a consideration in any interactive system. In domains where diagnosis rests on manual user observations, user interaction is unavoidable.

2.6 PROMISES AND PROBLEMS OF MODEL-BASED DIAGNOSIS

The model-based approach to diagnosis is a new direction compared to the traditional expert system approach. It has been claimed that it has the potential to overcome many of the limitations of the traditional systems, such as rule-based expert systems [45]:

- *Specific to one type of device*. Rule-based systems are usually tailored to a particular type of device.
- *Restricted to familiar devices*. The diagnosis knowledge is based on past experience of particular devices. If no experience has yet accumulated, it is not possible to build a rule-based system.
- *Restricted to known symptoms and faults*. Since the rules associate symptoms to faults, these must be known in advance. As soon as the system is faced with a problem that is not known, it fails completely. This is known as the "brittleness" of rule-based systems.
- *Unknown limits of competence*. It is usually difficult to assess (estimate) what the system can and can not diagnose, since the competence is encoded in a complex web of rules.
- *Not constructive*. There are no principles for automatically constructing diagnostic systems, and no principles for adapting to a new device.
- *Implicitness of the device structure*. There is no explicit representation of the structure (or function) of the device, since all knowledge is encoded in the rules.

These limitations suggest that it is doubtful if knowledge-based systems can be economically applied in a broader scope for technical diagnosis. In contrast to the rule-based systems, it is claimed that model-based diagnosis has a number of improvements [45]:

- *Explicit device structure*.
- *Constructive: based on domain specific model library.*
- *General: domain-independent diagnostic procedure*.
- *Diagnosis of new, unexperienced devices*.
- *Treatment of new symptoms and faults*, *e.g. multiple faults*.
- *Natural representation*.
- *Better characterization of the scope of competence*.

Other approaches to diagnosis than expert systems also suffer from the limitation that the diagnosis program is very specific to a particular device. In [16] it is claimed that:

> All classical approaches to diagnosis of artefacts, whether by means of FMECA (Failure Modes, Effects and Criticity Analysis), fault-trees or expert systems are based on the a priori identification of faults and failures, i.e., underlying disorders along with their externally visible symptoms, that can occur in the system to be diagnosed. Consequently, the program used to automate this process will be completely system-dependent, and any modification to this system will require a more or less complete rewriting of the program.

Next we discuss some problems of the model-based approach to diagnosis and consider alternatives by stating questions related to the purpose of the thesis.

Building the model One of the fundamental problems of model-based diagnosis is to construct the model of the domain. Some domains can be hard to understand and therefore difficult to model. For example, domains of circuits are often well-known, but a medical or mechanical domain can be substantially more complicated or even unknown in some parts. In [16] it is stated that modelling is the hard part of model-based diagnosis. Qualitative reasoning can be used either to reduce the complexity of the model or because there is no numeric model available. The idea is to model the function of the domain in terms of qualitative characteristics, such as {+, –, 0, ?} instead of quantitative (numerical) characteristics. By using qualitative reasoning techniques, the direction (e.g., +) of a quantity can be determined from the direction of its inputs. Is it an alternative to increase the level of abstraction in the model and represent some abstracted parts separately?

Completeness of the model Console et al. [11] identify a problem with the completeness of the domain model. In the consistency-based approach the completeness of the domain model is a common assumption. This has worked well in some domains such as diagnosis of electronic circuits, but is not adequate in medical diagnosis or mechanical troubleshooting since either a complete model is not available or is computationally intractable. There are ways of attacking this problem, such as working at multiple levels of abstraction to reduce the complexity. However, the lowest level of detail cannot be assumed to be complete, i.e. without abstractions. Console et al. suggest the use of incomplete causal models to address this problem. Is it possible to formulate some of the incomplete knowledge separately from the model and still integrate it in the reasoning process?

Failure probabilities Sherlock and GDE both use probabilistic information to guide the diagnosis process. In some domains this may be unknown or difficult to obtain. In a circuit domain the probabilities for component failure can be obtained from the manufacturer, but in other domains these probabilities would have to be collected from empirical sources—a task that may be very difficult. Can probabilities to some extent be replaced by focusing heuristics less detailed than probabilities for each component?

Integration of heuristic knowledge Heuristics can assist model-based diagnosis to generate candidates, focus on particular candidates, discriminate between candidates and help avoid measurements. How can heuristic knowledge be integrated with the knowledge in the model? Can heuristics be formulated in terms of the model to give semi-general knowledge?

Controlling the collection of observations In many diagnosis domains observations must be made by a human, not by a machine. In those cases user acceptance of the diagnosis process must be

taken into account. How can we control how the observations are requested by the system, so the user finds the sequence of questions natural?

User interface Observations plays a central role in diagnosis. For example, how can we ensure that the user correctly measures the right component? Can we take advantage of the model in designing the user interface, to support the user when the system asks for observations?

Imprecise measurements Measurements made by a user can be imprecise. Can a model be extended to reason with intervals, or can this be handled in some other way?

Chapter 3

Diagnosis in a Metalogic System

We discuss logic and metalogic as a representation language and outline a structure for a metalogic diagnosis system.

3.1 LOGIC KNOWLEDGE REPRESENTATION

Logic offers powerful possibilities to represent knowledge about a domain. Logic represents propositions about an external world as relations between individuals, expressed in sentences. The collection of sentences concerning a particular subject matter is a formal theory of that subject matter. Many parts of human knowledge have been represented as formal theories, such as parts of mathematics, geometry and physics. The main attractions of logic as a representation language are its semantics in the form of model theory, as developed by Tarski [47], and its ability to express very different subject matters with the same logical apparatus.

The subject matter of a domain can be viewed as an informal theory that can be formalized in logic as a formal theory. In formalizing an informal theory, we aim to generalize, simplify and to increase precision and rigour. We remove any meaning from the terms of the informal theory and express all conditions on their use explicitly in the sentences of the formal theory. Furthermore, we state explicitly the logical principles to be used in the deduction of theorems in the theory. When we have met these aims, the informal theory has been formalized—all content of the terms has been removed and only the form remains. For example, elementary number theory can be formalized as a formal theory **N**, see e.g. [34], where all the properties of the terms—e.g. ‘0 is a term’—are stated explicitly and all statements of the theory—e.g ‘s(0)+0 = s(0)’—can be deduced entirely by the rules stated explicitly in the theory. The subject matter of the domain can now be understood entirely by the standard semantics of predicate logic, no reference to the domain needs to be made in order to understand the expressions of the formal theory.

However, since the purpose of a formalization is to provide an exact characterization of an

informal theory, we can understand the formal theory in terms of the informal theory. Thus, a formal theory usually has an *intended interpretation*—that is, the terms of the formal theory can be interpreted as standing for objects of the informal theory and the predicates of the theory as representing relationships of the informal theory. For example, in the formal theory **N**, the intended interpretation of the symbol '0' is that it represents the concept of zero, and the intended interpretation of '+' is that it represents addition.

A formal theory is often stated in full first-order predicate logic. However, it can be stated in a subset of predicate logic—Horn clauses, if we do not need the full expressivity of predicate logic. In contrast to predicate logic, Horn clauses can be used as a programming language to mechanically derive logical consequences from the clauses. Prolog in [10] is an example of Horn clauses as a general logic programming language. The logical principle employed for Prolog systems is the resolution principle [42] augmented with the negation as failure rule, called SLDNF-resolution. The use of the negation as failure rule in a Horn clause theory can be justified with Clark's completed program [9]. See, for example, [36] for a tutorial of the semantics of logic programs. In this thesis we will use Horn clause theories and assume Clark's completed program for the clauses we give.

3.2 METALOGIC KNOWLEDGE REPRESENTATION

A metatheory is a theory for the study of another theory. That other theory is called the object theory because it is our object of study. The study is usually carried out in an informal manner. However, there is nothing that stops us from formalizing it as a formal metatheory. With a formal metatheory we can represent knowledge in the metatheory and relate this knowledge to the object theory.

The universe of discourse for a metatheory consists of formulas of an object theory, as opposed to an object theory, where the universe consists of objects of the world that you reason about. The universe of discourse for a metatheory may also involve other objects than object formulas. Since the world of a metatheory is made up of formulas of an object theory it is possible to relate one formula to another and to define when one follows from the other (or from a set of formulas.) With such a definition we replace a formal proof in the object theory with an informal proof in the metatheory. In this way we can discuss in the metatheory when sentences are true in the object theory. We can even skip intermediary steps in the proof of a sentence necessary to take if we would perform a formal proof in the object theory. Some of the formulas in these steps need not even exist as explicit formulas in the object theory—they are not necessary for the informal meta proof. Therefore, we see that the metatheory can simulate inferences in the object theory not possible within the object theory itself. The metatheory extends the set of inferences that are possible to make in the object theory. Therefore, the metatheory is capable of producing more powerful inferences than the object theory.

The capability to extend the set of inferences for an object theory is only one example of the expressive power that a metalogic formalization of knowledge makes possible. A meta theory can discuss properties of an object theory such as consistency, since an object theory is the object of study in a metatheory. A metatheory can thus handle and relate object theories to each other, which gives opportunities for non-monotonic reasoning and knowledge assimilation applications (see for example [35]).

3.3 A METALOGIC DIAGNOSIS SYSTEM

As discussed in the previous section it is possible to discuss formulas, theorems and other

properties of an object theory in a formal metatheory. We can handle different object theories and modifications of object theories as terms in the metatheory. This capability gives us a possibility to represent various kinds of reasoning as relationships in the metatheory.

In this thesis we investigate the prospects to represent a diagnosis system, which requires complex problem solving, as a metalogic representation structure. In Chapter 2 we discussed different approaches to diagnosis. One of the approaches was the so-called "model-based" approach which was contrasted with the rule based approach. The diagnosis system investigated in this thesis is an attempt to combine these two approaches by representing the different kinds of knowledge separately in the metalogic structure. Metalogic gives us a possibility to represent different kinds of knowledge separately, so that we can meet our goals for the representation of knowledge—retaining the logical structure and achieving a modular and transparent representation.

The model-based approach to diagnosis of a device is based on a comparison of an (incorrect) device and a representation of an ideal (correct) device. This approach requires a separation of the knowledge about each device. Fig. 3-1, below, illustrates how knowledge about the device that we would like to diagnose, and about an ideal device can be separated and related to each other as two logic theories. During the construction phase of a diagnosis system, an expert on the diagnosis area provides a conceptualization of the structure and function of a correct device. This conceptualization is formalized as an object theory, OT, itself an object of discourse in a metatheory, MT. The expert provides knowledge for the metatheory on forming hypotheses and conclusions about the device. In this theory heuristic hypotheses and conclusions about the system are defined in terms of the theorems of OT and knowledge about the device. The metatheory also formalizes a diagnostic strategy which studies the hypotheses and conclusions of MT in the diagnosis of the device. Knowledge about the real world, as perceived by the end user, is represented in MT. Consequently MT can consult the user about the device and collect information for the diagnosis.

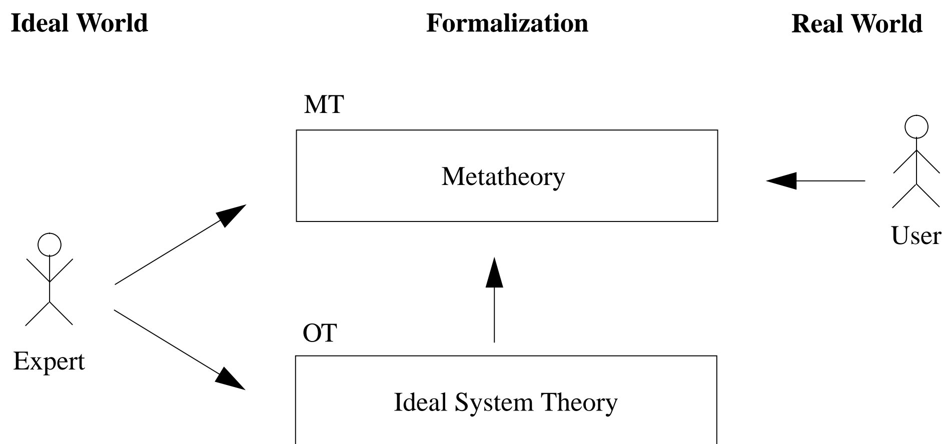


Fig. 3-1: *The structure of the diagnosis system.*

The arrows in Fig. 3-1 show where the knowledge for a theory comes from. Ideally, the construction process described above would be provided with computer support. In that case, the expert would be supported by a tool in which the expert would give a description of the device, partly graphically and partly by text, from which the tool generated the sentences of the theories. In Chapter 8 we will briefly discuss the prospects of providing computer support for the construction process.

In the next chapter we will discuss how we can represent knowledge about the structure and function of a domain as an object theory, whereas the metatheory will be discussed in Chapter 5.

Chapter 4

Formalization of a Diagnosis Domain

In this chapter we investigate the definition of a formal object theory for a diagnosis domain. The purpose is to build an explicit representation of a domain that can be used for diagnosis of a real-world situation when a fault has occurred. By comparing the statements of such a theory with observations in the real world it is possible to perform a diagnosis. Furthermore, the object theory will not contain any knowledge of diagnosis strategy—it will only contain a formalization of the structure and (correct) function of the domain. The diagnosis strategy will be represented as a separate (meta) theory, which will be discussed in the next chapter.

The kind of diagnosis domain that the theory will be designed to describe is a system of a set of components connected with connectors, through which some kind of directed flow is transported. The components display various observable behaviours and measurable flows. The system is controlled through various settings of the components.

As an example we discuss a stereo system with a set of components connected with cables. Although the example is comparatively small, a number of characteristics of a diagnosis domain are covered. The functionality of the stereo components is non-trivial, the system has different modes of operation, the flow transports music signals, the components display different behaviours and the system has different settings that affect the behaviours and the signals.

In Chapter 8 we discuss the prospects for adopting the diagnosis system to other domains. A domain with a similar structure as our example could be a monitoring and control system for a machine on an assembly line in industry.

4.1 AN EXAMPLE DOMAIN

An ordinary stereo system consists of an amplifier, some source components, two loudspeakers, cables for music signals and for power supply. A person can use the stereo for reproducing music from different source components, so the system can be used in different *modes*, e.g. the turntable mode. The user controls the system by adjusting controls on the stereo components, so depending on the

user's intention on what mode to use the system in, the user makes different *settings* of the components.

A central part of a stereo system is the amplifier to which all other components are connected. An amplifier has music input ports (Rec, Pb, etc.) and speaker output ports (A, B), as well as a mains power input port. Each port is dedicated for a certain music signal. In the figure below we see an illustration of an example amplifier which consists of music input and output ports, loudspeaker ports and a power input port.

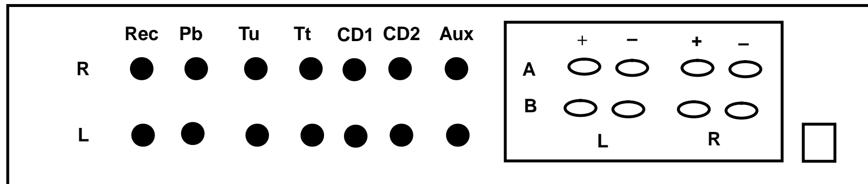


Fig. 4-1: *The back of an amplifier.*

We have two types of cables: music and power cables. A music cable is used to distribute the signals between stereo components whereas power cables are used to provide power to a component. A cable for a music signal usually consists of four pins—two for an input signal and two for an output signal where each signal has a positive and a negative polarity. An example music cable `w1` is illustrated in Fig. 4-2.

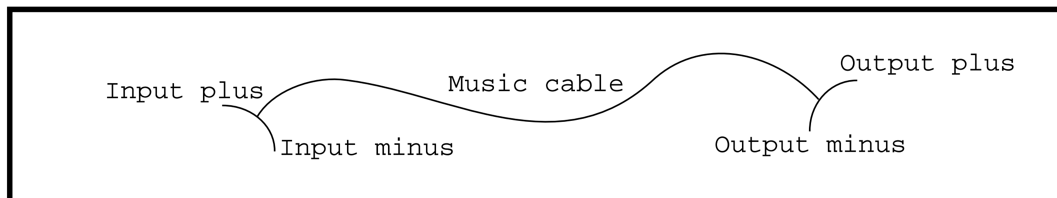


Fig. 4-2: *The subparts of a music cable.*

4.2 OVERVIEW OF THE OBJECT THEORY

The theory consists of two parts: (1) definitions of the structure of a system and (2) definitions of the function of a system. The structure definitions fall into two categories: (a) definitions of the objects of the system and (b) definitions of how the objects are connected to each other. Examples of the first category are stereo components and signal cables. An example of the second category is a connection between an amplifier and a cable for a loudspeaker, which consists of four simple connections—two for the loudspeaker cable and two for the amplifier. The main definition of function, `output`, builds on two definitions: `input` and `transformation`. The first characterizes the inputs to the objects and the second characterizes the measurable output signals and the observable behaviours of the objects in the domain. An output signal could be a voltage signal of an output port on an amplifier and a behaviour could be that the on-light of the amplifier is on.

4.3 STRUCTURE

The structure part of the theory falls into two parts—the objects and the configuration of the objects.

4.3.1 Objects

First we will define the objects of a system. We consider three types of objects: Components, connectors and external objects. Components are the nodes of a system that perform some kind of function—e.g. a CD-player which plays CD's. Connectors connect the nodes of a system and transport a flow of some kind–e.g. a music cable connected to the CD-player transports the source signals. External objects are not really part of the system but provide some essential service—e.g. a mains power outlet provides electricity.

The objects can be represented as lists. For example, the components `X` is a list where each element is a pair `[Type, Identity]`. This representation will help us to formulate compact heuristics in the metatheory. Since `Type` is a part of the object, a heuristic can be formulated in terms of the type of object, for example a loudspeaker type. For our stereo system example, we represent the objects of the system as shown below.

```
components(X) ←
     X = [ [amplifier,am],[tuner,tu1],[cassette_deck,ca1],
                [turntable,tt1],[cd_player,cd1],
                [loudspeaker,sp1],[loudspeaker,sp2]]

component(X1) ←
     components(X) & member(X1,X)

connectors(Y) ←
     Y = [[music_cable(cd_player),w1],[music_cable(turntable),w2],
        [music_cable(cassette_deck),w3],
        [music_cable(cassette_deck),w4],
        [music_cable(tuner),w5], [music_cable(loudspeaker),w6],
        [music_cable(loudspeaker),w7], [power_cable(amplifier),w8],
        [power_cable(cassette_deck),w9], [power_cable(tuner),w10],
        [power_cable(turntable),w11], [power_cable(cd_player),w12]]

connector(Y1) ←
     connectors(Y) & member(Y1,Y)

externals(E) ←
     E = [ [power,acdc(am)], [power,acdc(tu1)], [power,acdc(ca1)],
        [power,acdc(tt1)], [power,acdc(cd1)] ]

extern(E1) ← externals(E) & member(E1,E)

object(X) ← component(X) ∨ connector(X) ∨ extern(X)
```

We will now define the subparts for each type of object, to know what the objects consist of. The knowledge of how the objects are constructed is separate from the definitions of the objects. We can easily add objects to the theory by extending a list of objects. For example, we can add a second CD-

player `cd2` to the theory by adding it to the list of `components`, without again specifying the subparts of a CD-player.

The interior structure of the objects will not be described—only the exterior subparts of the objects. The reason is that we choose a level of abstraction that does not describe the interior of the objects. The stereo system is modelled at the level of stereo components—not at the level of the circuits of the components.

Below we find the definition of `subparts` for all objects in the domain. In this definition `S` are the subparts of a stereo object `O`. Each subpart in this definition has the same structure—a pair `[Type, Identity]`. For example, `[mu_port, in(cd1/r,N)]` is a music port with the identity `in(cd1/r,N)`, which indicates the right channel CD1 input port on an object identified by `N`. Using the identity, `N`, of the component as part of the identity of the subparts gives us some generality in the representation, since the same definition holds for all components of a particular type, e.g., for all CD-players.

```
subparts(O,S) ←
        O = [amplifier,N] &
        S = [[mu_port,out(rec/r,N)],[mu_port,out(rec/l,N)],
                [mu_port,in(pb/r,N)],[mu_port,in(pb/l,N)],
                [mu_port,in(tu/r,N)],[mu_port,in(tu/l,N)],
                [mu_port,in(tt/r,N)],[mu_port,in(tt/l,N)],
                [mu_port,in(cd1/r,N)],[mu_port,in(cd1/l,N)],
                [mu_port,in(cd2/r,N)],[mu_port,in(cd2/l,N)],
                [mu_port,in(aux/r,N)],[mu_port,in(aux/l,N)],
                [sp_port,out(a/l/pl,N)],[sp_port,out(a/l/mi,N)],
                [sp_port,out(a/r/pl,N)],[sp_port,out(a/r/mi,N)],
                [sp_port,out(b/l/pl,N)],[sp_port,out(b/l/mi,N)],
                [sp_port,out(b/r/pl,N)],[sp_port,out(b/r/mi,N)],
                [power,inv(220,N)],[light,on(N)],[button,volume(N)],
                [button,speaker(N)],
                [button,on(N)],[button,tuner(N)],
                [button,cassette_deck(N)],[button,turntable(N)],
                [button,cd_player(cd1,N)],[button,cd_player(cd2,N)]]

subparts(O,S) ←
        O = [tuner,N] &
        S = [[mu_port,out(tu/r,N)],[mu_port,out(tu/l,N)],
                [power,inv(220,N)],[light,on(N)],[button,on(N)]]

subparts(O,S) ←
        O = [cassette_deck,N] &
        S = [[mu_port,out(pb/r,N)],[mu_port,out(pb/l,N)],
                [mu_port,in(rec/r,N)],[mu_port,in(rec/l,N)],
                [power,inv(220,N)],[light,on(N)],[button,on(N)]]
```

```
subparts(O,S) ←
        O = [turntable,N] &
        S = [[mu_port,out(tt/rl,N)],[power,inv(220,N)],
                [light,on(N)],[button,on(N)]]

subparts(O,S) ←
        O = [cd_player,N] &
        S = [[mu_port,out(cd/rl,N)],[power,inv(220,N)],
        [light,on(N)],[button,on(N)]]

subparts(O,S) ←
        O = [loudspeaker,N] &
        S = [[sp_port,in(pl,N)], [sp_port,in(mi,N)]]

subparts(O,S) ←
        O = [music_cable(T),N] & (T = cd_player ∨ T = turntable) &
        S = [[pin,in(pl,N)], [pin,in(mi,N)], [pin,out(plmi,N)]]

subparts(O,S) ←
        O = [music_cable(T),N] &
        (T = cassette_deck ∨ T = loudspeaker ∨ T = tuner) &
        S = [[pin,in(pl,N)], [pin,in(mi,N)],
        [pin,out(pl,N)], [pin,out(mi,N)] ]

subparts(O,S) ←
        O = [power_cable(T),N] &
        S = [[plug,inv(220,N)], [plug,outv(220,N)]]
```

4.3.2 Configuration

We have defined the objects of a system—components, connectors and external objects—now we need to describe how they are connected. The objects can be connected in different ways—different system configurations. For example, the loudspeakers in a stereo system may be connected in different ways to the amplifier. We can represent this in `configuration` as a relation between a system `Sy` and a configuration `C` of components, connectors and external objects. A configuration is a list where each element is a structure:

```
Object1 -- Connections1 -- Connector -- Connections2 -- Object2
```

such that `Object1` and `Object2` are connected via `Connector` in the system. The details of the connection between `Object1` and `Connector` are given by `Connections1`, likewise for `Object2`, `connector` and `connections2`. A single element of a configuration is a `configuration_unit`. Below we find a part of `configuration` for an example stereo system `s1`.

```
configuration(SY,C) ← SY = s1 &
     C = [ [cd_player,cd1] --
                 [ [mu_port,out(cd/rl,cd1)] -- [pin,out(plmi,w1)] ] --
        [music_cable(cd_player),w1] --
                 [ [pin,in(pl,w1)] -- [mu_port,in(cd/r,am)],
                  [pin,in(mi,w1)] -- [mu_port,in(cd/l,am)] ] --
        [amplifier,am],

        [tuner,tu1] --
                 [[mu_port,out(tu/r,tu1)] -- [pin,out(pl,w5)],
                  [mu_port,out(tu/l,tu1)] -- [pin,out(pl,w5)] ] --
        [music_cable(tuner),w5] --
                 [[pin,in(pl,w5)] -- [mu_port,in(tu/r,am)] ,
                  [pin,in(mi,w5)] -- [mu_port,in(tu/l,am)] ] --
        [amplifier,am] ,

        ... ]
```

The ellipsis (...) indicates that only a part of the configuration relation is given. This provides an overview of how a configuration is represented, but the details of the representation are not essential for an understanding of the object theory.

```
configuration_unit(Sy, X1--UV1--Y-UV2--X2) ←
     configuration(Sy, C) &
     member(X1--UV1--Y-UV2--X2, C)
```

4.4 FUNCTION

The second part of the object theory is a formalization of the function of a system. We want to know how the system works in a particular situation. The theory will give us statements of the signals and behaviours of a system, which we can compare with observations of a system in the real world to make a diagnosis. For example, when a CD-player operates it produces music signals on its output ports and there will be a light on the front panel to indicate that it is on.

4.4.1 Output

To represent the function of a system we define a relation `output` between an object `X`, a system use `U` and a list `Out` of output signals and behaviours. `U` is a list `[Sy,M,S]`, where `Sy` is the system configuration, `S` is a list of settings for the system and `M` is a mode of use. The relation is true when there is a list of inputs `I` to `X` that can be transformed to a list `Out` of outputs.

```
output(X,U,Out) ←
        U = [Sy,M,S] & object(X) &
        input(X,U,I) & transformation(I,X,U,Out)
```

Example:

In a stereo system s1, what would the outputs be from an amplifier am when the system is used in the CD-player mode? Let X be [amplifier,am], U be [s1,cd_player,S], where S is as below where the first of the settings is that the CD-player button of the amplifier is on.

```
S = [   setting(s1, [amplifier,am], cd_player, [button,cd_player(cd1,am)], on),
        setting(s1, [amplifier,am], cd_player, [button,on(am)], on),
        setting(s1, [amplifier,am], cd_player, [button,speaker(a,am)], on),
        setting(s1, [cd_player,cd1], cd_player, [button,on(cd1)], on)        ]
```

Then, Out would be as below. The output signals would be values for the speaker ports of the amplifier. The theory would assert 66W signals on the amplifier's loudspeaker ports. The asserted behaviour would be that the on-lights of the amplifier and the CD-player are on.

```
Out = [ signal(s1, [amplifier,am], cd_player, [sp_port,out(a/l/pl,am)],66 w),
        signal(s1, [amplifier,am], cd_player, [sp_port,out(a/l/mi,am)],66 w),
        signal(s1, [amplifier,am], cd_player, [sp_port,out(a/r/pl,am)],66 w),
        signal(s1, [amplifier,am], cd_player, [sp_port,out(a/r/mi,am)],66 w),
        behaviour(s1, [amplifier,am], cd_player, [light,on(am)], on),
        behaviour(s1, [cd_player,cd1], cd_player, [light,on(cd1)], on)    ]
```

End of example.

We discuss the definitions of input and transformation below, to explain how the above values are derived.

4.4.2 Input

The input signals to an object depend on the configuration of the system and the settings of the objects. For example, the cable connections of a stereo system determine what inputs the stereo components receive; the settings of an amplifier determine what inputs a loudspeaker receives. We can represent this as a relation input between an object X, a system use U and a list I of inputs. We state a clause for each type of object in the definition. We will use the configuration_unit relation to determine which object is connected to which.

Below we find a definition of input for the stereo system example. An an illustration, consider the second clause of input. This clause is for an amplifier, which has two kinds of inputs. The first kind is determined by the setting of a source button (CD, Cassette, etc.) on the amplifier. The outputs of the cable connected to the component indicated by that setting (e.g. cd1), are propagated to the input ports of the amplifier, according to the connections between the cable and the amplifier. The second kind of input is the power input signal, which also can be determined by the configuration and propagated correspondingly.

```
input(X,U,I) ←
        U = [Sy,M,S] &
        X = [loudspeaker,N] &
        configuration_unit(St, X2--Conns1--Y--Conns2--X) &
        output(Y,U,Out) &
        propagate(Out,Conns2,I)

input(X,U,I) ←
        U = [Sy,M,S] &
        X = [amplifier,N] &
        setting(setting(Sy,X,M,[button,M(N2,N)],on), X, M, S) &
        component(X2) & X2 = [M,N2] &
        configuration_unit(Sy, X2--Conns1--Y1--Conns2--X) &
        output(Y1,U,Out1) &
        propagate(Out1,Conns2,I1) &
        configuration_unit(Sy, [[power,acdc(N)]--Conns3--Y2--Conns4--X]) &
        output(Y2,U,Out2) &
        propagate(Out2,Conns4,I2) &
        append(I1,I2,I)

input(X,U,I) ←
        U = [Sy,M,S] &
        X = [T,N] & member(T,[tuner,cassette_deck,turntable,cd_player]) &
        configuration_unit(Sy,[power,acdc(N)]--Conns1--Y--Conns2--X) &
        output(Y,U,Out) &
        propagate(Out,Conns2,I)

input(X,U,I) ←
        U = [Sy,M,S] &
        connector(X) &
        configuration_unit(Sy,X2--Conns1--X--Conns2--X3) &
        output(X2,U,Out) &
        propagate(Out,Conns1,I)

input(X,U,I) ←
        U = [Sy,M,S] &
        X = [power,acdc(N)] &
        I = [signal(Sy,X,M,N,220 v)]
```

As discussed above, the settings affect the function of a component. We capture this in the definition of `input` where the system use `U` involves a list `S` of settings for the system. However, we can represent a set of default settings, to be used if settings needed to determine the function of a component are not given in `S`. These are stated in the relation `standard_settings`. So we state that `S1` is a setting, either if it is in a list `S` of settings, or if it is a standard setting for the object `X` in mode `M`.

```
setting(S1,X,M,S) ←
     member(S1,S) & S1 = setting(_,_,_,_)
setting(S1,X,M,S) ←
     ¬ member(S1,S) & standard_setting(X,M,S1)

standard_setting(X,M,S1) ←
     standard_settings(X,M,S) & member(S1,S)
```

Below we find the standard settings defined for the stereo system example, one of which is the volume setting of the amplifier.

```
standard_settings(X,M,S) ←
     X = [amplifier,N] &
     S = [ setting(Sy,X,M,[button,volume(N)],30),
        setting(Sy,X,M,[button,speaker(a,N)],on),
        setting(Sy,X,M,[button,on(N)],on),
        Setting(SY,X,M,[button,M(N)],on) ]

standard_settings(X,M,S) ←
      X = [T,N] & member(T,[tuner,cassette_deck,turntable,cd_player]) &
      S = [setting(Sy,X,M,[button,on(N)],on) ]
```

4.4.3 Transformation

As defined above, we represent an output as an input that can be *transformed*. We must determine the transformation rules for the components in order to represent the function of a system correctly. Transformation of input signals to output signals and behaviours can be represented as a relation `transformation` between a list `I` of inputs, an object `X`, a system use `U`, and a list `Out` of output signals and behaviours. We will state a clause for each type of object that has its own kind of transformation rules for its input signals. The `out_signal` relation will state transformation rules for subparts of objects.

For example, the transformation rules for an amplifier are stated over its output speaker ports. The setting of the speaker button determines which set of speaker ports (A or B) will have output signals. The predicates `out_signals` and `out_signal` in turn specify the transformation rules for these ports.

```
transformation(I,X,U,Out) ←
              U=[Sy,M,S] &
              X = [loudspeaker,N] &
              setting(setting(Sy,X,M,volume,Vol), X, M, S) &
              sum_input_signals(I,Sum) &
              calculate((Sum * Vol)/50, Db) &
              (Sum > 15 & T=good ∨ Sum ≤15 & Sum > 0 & T=bad ∨ Sum=0 & T=none) &
              Out = [signal(Sy,X,M,volume,db),
                     behaviour(Sy,X,M,sound,T) ]

transformation(I,X,U,Out) ←
              U=[Sy,M,S] &
              X = [amplifier,N] &
              setting(setting(Sy,X,M,[button,speaker(N)],Sp), X, M, S) &
              speaker_ports(P,X,Sp) &
              out_signals(P,X,I,U,Out1) &
              Out = [behaviour(Sy,X,M,[light,on(N)],on) | Out1]

transformation(I,X,U,Out) ←
              U=[Sy,M,S] &
              X = [T,N] & member(T,[tuner,cassette_deck,turntable,cd_player]) &
              out_music_ports(P,X) &
              out_signals(P,X,I,U,Out1) &
              Out = [behaviour(Sy,X,M,[light,on(N)],on) | Out1]

transformation(I,X,U,Out) ←
              U=[Sy,M,S] &
              connector(X) &
              output_pins(P,X) &
              out_signals(P,X,I,U,Out)

transformation(I,X,U,Out) ←
              U=[Sy,M,S] &
              X = [power,acdc(N)] &
              out_signal(signal(Sy,X,M,N,Q), I) &
              Out = [signal(Sy,X,M,N,Q)]
```

Finally, we need to specify transformation rules of signals for ports of components and pins of connectors. We define a relation `out_signals` between a list `P` of ports (or pins), a component `X`, a list of inputs `I`, a system use `U`, and a list `Out` of output signals for the elements in `P`.

```
out_signals(P,X,I,U,Out) ←
     empty(P) & empty(Out)
out_signals(P,X,I,U,Out) ←
     P = [P1|PW] &
     U = [Sy,M,S] &
     O = signal(Sy,X,M,P1,Q) &
     out_signal(O,I) &
     out_signals(PW,OutW) &
     Out = [O|OutW]
```

```
out_signal(O,I) ←
        O = signal(Sy,X,M,P,Q) & X = [T,N] &
        P = [sp_port,out(AB/LR/PM,N)] &
        member(signal(Sy,X,M,[power,inv(220,N)],220 v), I) &
        member(signal(Sy,X,M,[mu_port,in(N2/r,N)],K mv), I) &
        member(signal(Sy,X,M,[mu_port,in(N2/l,N)],K2 mv), I) &
        component([M,N2]) &
        calculate( (K1+K2)/12.12, K) &
        Q = (K w)
out_signal(O,I) ←
        O = signal(Sy,X,M,P,Q) & X = [T,N] &
        P = [mu_port,out(N2/RL,N)] &
        member(signal(Sy,X,M,[power,inv(220,N)],220 v), I) &
        Q = (400 mv)
out_signal(O,I) ←
        O = signal(Sy,X,M,P,Q) &
        connector(X) & subpart(P,X) &
        partner_in_connector(P,P1,X) &
        member(signal(Sy,X,M,P1,Q), I)
out_signal(O,I) ←
        O = signal(Sy,X,M,P,Q) &
        X = [power,acdc(N2)] &
        member(signal(Sy,X,M,N2,Q), I)
```

Note that `transformation` only specifies a rule, `out_signal`, for the transformation of the input signals to an object—it does not specify the internal function of the object in detail. The reason is a desire to keep the level of detail low in the object theory. Two arguments can be given for why a highly detailed theory should be avoided. The first is that the computation involved for drawing conclusions from the theory will be less complex with a less detailed theory, and the second is that it will be easier for the designer of a theory to describe the function of a domain with a less detailed theory. We will discuss this issue further in section 5.4.1.

4.5 DERIVATION FROM THE THEORY

We have investigated the definition of a formal object theory for a system of a set of components connected with connectors. To exemplify such a theory we discussed a theory of the structure and function of a simple stereo system. This theory was formulated in Horn clauses which can be translated to the syntax of existing logic programming languages. For instance to translate to Prolog, we replace the implication symbol by ':-', the conjunction symbol by ',', the disjunction symbol by ';', the negation symbol by 'not'[1] and insert a full stop at the end of each clause.

When we have translated our object theory to a logic program we can check the "correctness" of our theory. This would be done by deriving the outputs for every object and comparing them with a real correct stereo system. This way we can check if we have made any mistakes in our formalization. For example, the following answers can be derived from the logic program, when implemented in a Prolog system (`?-` indicates a query to the Prolog system). See also the example on page 25.

Example:
1. What are the subparts of a tuner?

```
?- X = [tuner,tu1], component(X), subparts(X,L).

L = [[mu_port,out(tu/r,tu1)],[mu_port,out(tu/l,tu1)],
     [power,inv(220,tu1)],[light,on(tu1)],[button,on(tu1)]]
```

2. What settings and outputs does a normal tuner have?

```
?- X = [tuner,tu1], U = [s1,tuner,S], output(X,U,Out).

S = [setting(s1,[tuner,tu1],tuner,[button,on(tu1)],on)]
Out = [signal(s1,[tuner,tu1],tuner,[mu_port,out(tu/r,tu1)], 400 mv),
        signal(s1,[tuner,tu1],tuner,[mu_port,out(tu/l,tu1)], 400 mv),
        behaviour(s1,[tuner,tu1],tuner,[light,on(tu1)],on) ]
```

In a diagnosis system, we will deduce consequences of the theory to diagnose malfunctions. We represent only correct function and behaviour in the object theory, whereas the metatheory can contain knowledge about faulty function and behaviour. Therefore, all consequences of the object theory should be observable in the real world and any that is not, would indicate a fault. The diagnosis of such faults will be the subject of the next chapter.

---

1. Note that Prolog imitates logical negation with the negation as failure rule, as discussed in section 3.2. This may give unsound results if the computation rule is not safe, i.e. if a negative literal that is not ground can be selected for resolution.

Chapter 5

A Diagnosis Theory

In this chapter we investigate the prospects of giving a representation of a diagnosis strategy that is separated from the knowledge of a particular domain. We aim to give an example of a diagnosis strategy. The separation of the representation of a diagnosis strategy from the representation of a diagnosis domain is motivated by reasons of transparency, modularity and the logical structure of diagnosis knowledge, as discussed in Chapter 1. In Chapter 4 we took the first step towards this separation of knowledge by giving a theory for a diagnosis domain. The diagnosis domain was exemplified by a stereo system. As the second step we define another theory: a theory for the diagnosis strategy. We will refer to this theory as the *metatheory*, and the theory for the diagnosis domain as the *object theory*. The object theory thus contains domain knowledge (the structure and function of the diagnosis domain), whereas the metatheory contains knowledge about the diagnosis strategy. The latter is called the metatheory since it discusses sentences of the object theory. By using two separate theories we aim to achieve our goals for the representation of diagnosis knowledge: (1) logical structure: the two-level structure of the knowledge can be retained when domain and diagnosis knowledge are separated into two theories that have a two-level relationship (object – meta), (2) modularity: we can modify the object theory if the diagnosis domain changes without modifying the metatheory, or we can alter the metatheory for a new diagnosis strategy without changing the object theory, and (3) transparency: with two theories we do not have to mix knowledge of different kinds, instead we can represent the different kinds clearly apart, promoting transparency of knowledge.

Knowledge of different kinds that is represented separately is useful only if it can work together to effectively solve problems. The separation of knowledge into two logic theories would be of little interest if there were no possibilities to make them work together. Fortunately, metalogic provides the means for this through the meta–object relationship of the theories—the metatheory takes sentences of the object theory as objects of discourse, in the same manner as the object theory takes objects of the diagnosis domain as objects of discourse. We see this ability of metalogic to discuss the sentences of a theory in a metatheory as a key to making knowledge of different kinds work together.

In Chapter 2 we identified five parts of the diagnosis process that we regard as central and want to represent in the metatheory. These parts were: the handling of observations, the generation of hypotheses, the refutation of hypotheses, the ordering of the hypotheses to refute and handling of the user interaction. The first four parts of the representation in the metatheory will be discussed in this chapter. The user interaction will be performed by a high-level interface, for example, by using a graphical display of the diagnosis domain, as discussed in Chapter 7.

5.1 OBSERVATIONS

The observations in a diagnosis domain offered by the user of the diagnosis system are central in a representation of a diagnosis strategy, because the reports provide the basis for the investigation of the hypotheses. When the diagnosis system investigates the hypotheses, observations of the diagnosis domain must be available so the system can relate the observations with the hypotheses and draw conclusions about the diagnosis case. In the following, we shall identify what types and characteristics of observations we would like to capture in our representation.

Some observations made by the user apply only to a specific mode of use of a system, whereas other observations apply to all modes. For example, in a stereo system let us assume that a CD-player is faulty and causes the loudspeaker's sound to deteriorate. An observation that a loudspeaker sounds bad would then apply to the CD-player mode but not to the turntable mode, whereas an observation that two components are connected would apply equally well to both modes. We thus need to indicate, for some of the observations, in which mode the observation is made by the user.

A system is controlled by setting controls of the components, therefore settings are interesting observations, for example in a stereo system,

Some observations require some kind of device to measure a value, such as the voltage of a component (a signal), whereas some observations can be made directly by the senses, for example that a light is on or off (a behaviour). This difference between kinds of observations can be important when the diagnosis system decides which questions to ask the user. In consequence, we should specialize observations with respect to the mode of use for the diagnosis domain, and also specialize observations with respect to how the parameter can be observed. Therefore we distinguish four kinds of observations—observations of connections, settings, signals and behaviour.

Below we define four kinds of observation terms to represent the four kinds of observations that we want to distinguish. We choose to handle the observations in the metatheory because they represent what is known about the diagnosis domain in the real world—we do not include them in the object theory which represents the ideal world.

Definition 5-1

An *observation* is a term that has one of the following forms:

signal(System, Object, Mode, Type, Value)
behaviour(System, Object, Mode, Type, Value)
setting(System, Object, Mode, Type, Value)
conn(System, Obj1, Obj2)

where *System* represents a system configuration, *Object* an object in the diagnosis domain, *Mode* the mode of use for the diagnosis domain, *Type* a type specification of *Value*, which is a simple value. *Obj1* and *Obj2* represent connected objects in the domain. These observation terms should be seen as a basic

set of terms for the representation of user reports about a diagnosis domain, satisfactory for our example. Other domains, or a more complex stereo system, might require more complex or more vague descriptions of observations. For instance, the sound of a loudspeaker could be described in more elaborate terms than *bad*.

Example:

The user might observe, first, that the loudspeaker *sp1* sounds bad when the stereo system is used in CD-player mode and, second, that the output port *p7* on loudspeaker *sp1* has a value of 400 mV. These two facts are individually represented as two metatheory terms. See Fig. 5-1 below for an illustration of the terms.

behaviour(s1, [loudspeaker, sp1], cd_player, sound, bad)
signal(s1, [loudspeaker, sp1], cd_player, [port, sp1(p7)], 400 mV)

Fig. 5-1: *Two observations represented as terms.*

Note that in observations the terms are names for objects of the object theory, e.g. *[loudspeaker, sp1]* is a metalevel name for the object level term `[loudspeaker, sp1]`. If we represent observations as terms in a metatheory, the terms can be collected in, e.g., a list which represents the observations made by a user. The observations of the user are thus handled as objects in the metatheory, which gives a foundation for various kinds of reasoning. For instance, we can say what conclusions can be drawn from a specific set of observations, but also classify observations, e.g. the degree of uncertainty of some observation, the time of observations or make other classifications. We shall have more to say on the relationship between object and metalanguage in the following section.

5.2 HYPOTHESES

We now discuss how we can view hypotheses and how they relate to observations. When making a diagnosis an expert will often start by asking some initial questions in order to generate a set of hypotheses, which are then tested by asking further questions and/or performing tests of various kinds. Therefore it seems natural to represent a method for generating hypotheses in the strategy of a diagnosis system. We can view a hypothesis as pointing out a problem area of the diagnosis domain. A problem area identifies an incorrect property, or class of properties, of an object, or class of objects, in the diagnosis domain. The settings of a component and the internal function of the loudspeakers are examples of hypotheses pointing out problem areas. By distinguishing areas of the domain we can give the diagnosis strategy the ability to reason with something more comprehensive than single parts of the diagnosis domain. In a diagnosis process it is convenient to refer to a problem area rather than to single suspects (one property of a component that is suspected to be faulty), because heuristics can reason in terms of these areas rather than in terms of single suspects and the generation of hypotheses can be done in terms of them. In order to let the strategy reason in terms of hypotheses, we represent the hypotheses as objects in the metatheory.

5.2.1 Abnormal Observations

The starting point for a diagnosis is a report, from the user, of a fault in the diagnosis domain. So, we must have some way of dealing with faults to discuss and represent them in our metatheory. Since we have a theory of the diagnosis domain in the form of an object theory, we find it convenient to define faults relative the object theory. Everything that can be deduced from an object theory can be considered

as correct (for that theory), since it represents the ideal world for a particular diagnosis domain. Consonant with this, statements that contradict what can be deduced from the object theory, can be seen as faults (for the domain that the theory formalizes). We take this view as a starting point to classify the observation terms of Definition 5-1 in the metatheory.

We call observations that correspond to certain theorems of the object theory for *normal* observations, and, consequently, those that are not normal for *abnormal* observations. The object theory given in the previous chapter formalizes the function and structure of a domain in the predicates *output* and *configuration*, respectively. Thus, the correspondence between observation terms and theorems of the object theory should be defined over these predicates, as equality of their terms with observation terms. A characterization of ‘equal’ needs to be given for every particular domain to account for variations of e.g. measurement devices. For instance, a measurement of 390 mV or 410 mV observed by the user might be accepted to be the same as 400 mV defined in the predicate *output* in the object theory. This characterization can be given in the object theory or in the metatheory, but since the accepted variation of values are presumably different in different domains, it seems natural to put it in the object theory to keep the metatheory more general. Furthermore, equal is characterized as a three-place relation between two values and a type, to provide different definitions for different types of values. An alternative would be to reason with intervals, i.e. propagate intervals in the theory, as in SOPHIE [3]. We now give definitions for normal and abnormal observations.

Definition 5-2

A *normal observation O* for an object theory *T*, is an observation term that corresponds to a theorem of the predicates `output` or `configuration` of *T*.

An *abnormal observation O* for an object theory *T*, is an observation term and does not correspond to a theorem of the predicates `output` or `configuration` of *T*.

Following Definition 5-1 and Definition 5-2 we can give logic programs for *normal* and *abnormal* as relations between a theory and an observation term. Below we give the first clause for each program, which cover the observation term *signal*. Each clause covers one kind of the observation terms *signal*, *behaviour*, *setting* and *conn*.

normal(O, T) ←
    O = signal(System, Object, Mode, Type, Value1) &
    demo(T, output(Object, [System,Mode,Settings], Out)) & **(5-1)**
    member(signal(System, Object, Mode, Type, Value2), Out) &
    demo(T, equal(Value1, Value2, Type))

abnormal(O, T) ←
    O = signal(System, Object, Mode, Type, Value1) &
    demo(T, output(Object, [System,Mode,Settings], Out)) & **(5-2)**
    ¬ (member(signal(System, Object, Mode, Type, Value2), Out) &
        demo(T, equal(Value1, Value2, Type)))

In the second conjunct, *demo*, of these programs, *output(Object, [System, Mode, Settings], Out)*, is a partial name (containing metavariables) for the object theory formula
`∃Object∃System∃Mode∃Settings∃Out(output(Object,[System,Mode,Settings], Out) ).`

Likewise is *equal(Value1, Value2, Type)* a partial name for the corresponding object theory formula.

The definitions of *normal* and *abnormal* derive their observation terms from an object theory. We thus see that we need a demo predicate to relate a theory and a partial name for an object level formula (a non-ground metalevel term). The predicate is true when the formula named by the partial name can be derived from the theory. The demo predicate was originally proposed by Bowen and Kowalski [2] as a representation of the derivability relation in an object language, part of an amalgamation of the object language and the metalanguage. Various approaches to the problem of relating object language and metalanguage have been studied, for example FOL in [50], Gödel [30] and Reflective Prolog in [14], but at present it seems that no solution has yet been generally agreed upon. In this thesis we only indicate two general approaches to the problem.

The first approach is to implement the demo predicate in Prolog as an ordinary meta-interpreter that takes advantage of Prolog's unification for handling the variables in the partial names. With this simple approach, we would mix object language and metalanguage and risk confusion between metalevel variables and object level variables, but if we were careful not to use the same variable names for object level and metalevel variables we could avoid such confusion. However, the semantic mix of languages would still remain, making this approach less appealing.

An alternative approach would be to separate the object and metalevel variables by using a ground representation of object level terms in the metatheory. All object level constants could name themselves autonymously [4]—the metalevel term *cd_player* would name the object level constant `cd_player`, whereas object level variables would be named by ground metalevel terms. For instance, the ground metalevel term *?a* could name the object level variable `A`. The metatheory names would need to be converted to object theory formulas by the theorem prover, when executing the demo predicate. That is to say, if a name is partial (contains metavariables) it must be converted to an object level formula with object level variables in those places where there are metavariables in the partial name. After the execution of the demo predicate, the object level formula, with substitutions, must be converted back to a (ground or non-ground) metalevel term. For this purpose we would need a naming relation between metatheory names for object theory formulas and object theory formulas.

A naming relation should be part of a meta-metatheory since the theory must be capable of handling both the object language and the metalanguage. It is not advisable to define a naming relation between object language expressions and metalanguage expressions in the metalanguage itself, because this would lead to confusion as to whether an expression is a term or a formula. Consider the following example of the use of a naming relation defined in the metalanguage:

$$name(p(?x), Form) \ \& \ Form = p(X) \ \& \ p(X)$$

where the first occurrence of *p(X)* is a *term* of the metalanguage and the second occurrence a *formula* of the metalanguage. This naming relation would give us a semantically closed metalanguage in which it would be possible to refer to its own expressions.

$$true(p(?x)) \leftarrow \ name(p(?x), P) \ \& \ \neg P$$

In this thesis the meta-metatheory is informal and we will not give a formalization of the naming relation since the problem is beyond the scope of the present work. However, in [1] we discuss an implementation of a naming relation in a meta-metatheory. This implementation is a ground representation approach, in accordance with the alternative approach discussed above.

To conclude our discussion of normal and abnormal observations, we see that all observation terms corresponding to theorems of the object theory predicates `output` and `configuration` are normal observations. As a consequence of the definition of normal and abnormal observations, the definitions are modular. If we modify the object theory, the definitions of *normal* and *abnormal* will not have to be altered since they are defined relative an object theory. If the object theory is modified the *abnormal* and *normal* metatheory predicates will derive different observations without any changes in the metatheory. With these definitions, it is also possible, as we will see later, to reason that what is abnormal in one theory can be normal in another theory, since normal and abnormal are defined relative a particular theory. A fault can therefore be viewed as an abnormal observation. The user observation that initiates the diagnosis process is referred to as the selected abnormality. This is the basis for the generation of hypotheses in a diagnosis case in relation to an object theory, as will be discussed in the next section.

5.2.2 The Signal Graph

We can take advantage of the object theory when we generate hypotheses for a fault (an abnormal observation) reported by the user. The formalization tells us which objects influence the output observed by the user, so by studying the object theory hypotheses for the fault can be generated. We therefore analyse the object theory to see what could have caused the fault. To aid the analysis we use the concept of signal path. This can be understood as a sequence of objects that a signal traverses in order to construct an output from an object in the diagnosis domain, such as a sound from a loudspeaker. We see the concept of signal paths applicable in a class of domains. As will be discussed in Chapter 8, the stereo object theory can be viewed as an instance of an abstract theory for a class of diagnosis domains—the class that consists of objects connected together with some type of directed flow between them. In this class it should be possible to analyse the paths the signals between the objects take for each output of the objects. Furthermore, it is possible to view these paths as a directed acyclic graph since the signals are going in a certain direction, and the signals do not form any cycles. Each combination of object and mode in the diagnosis domain thus defines a particular *signal graph*. For example, the output of the loudspeaker port on the amplifier in cd-player mode has a graph that is different from the output of the on-light on the cd-player, which both are different from the graph for the output of the loudspeaker port on the amplifier when it is used in the turntable mode. Therefore, there can be many possible signal graphs for one diagnosis domain. Since the signal graph tells us what objects are involved for each output, it is a convenient tool in the diagnosis of faulty outputs. Note that the signal graph is not represented explicitly in the object theory—it is constructed in the metatheory by analysing sentences of the object theory. This construction is made during the diagnosis. It would be possible to 'pre-compile' the signal graphs of a given object theory before the diagnosis, but any modification of the object theory during the diagnosis affecting the signal paths would then necessitate the system to reconstruct the signal graphs during diagnosis.

The signal graph is a directed acyclic graph (DAG) of nodes where the nodes correspond to objects and the arcs correspond to signal flows of the diagnosis domain. In this DAG the root is the component that displays the fault reported by the user, and the leaves are the components that do not

have any input signals. As an example, consider the graph for a component *loudspeaker1* when a stereo system is used in cd-player mode, in Fig. 5-1.

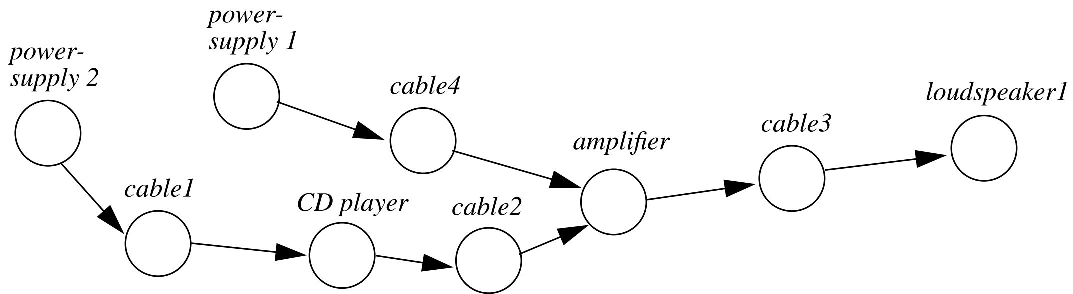


Fig. 5-2: *A signal graph*

As the figure indicates, the paths for the loudspeaker when the stereo system is used in cd-player mode are (1) power-supply-1 to cable4 to amplifier to cable3 to loudspeaker1, (2) power-supply-2 to cable1 to cd-player to cable2 to amplifier to cable3 to loudspeaker1. These paths form a directed acyclic graph, which we can represent as a term in the metatheory. See Fig. 5-1 below.

d([loudspeaker1, cable3, amplifier,
d([[cable4, power_supply1], [cable2, cd_player, cable1, power_supply2]])])

Fig. 5-3: *A signal graph represented as a term.*

The signal graph for an output is constructed by a metalevel analysis of the object theory. We have designed a meta-interpreter *solve_construct* for the analysis of an object theory. A standard meta-interpreter can be specialized to construct the signal graph, by using knowledge about the structure of the object theory. The knowledge needed is that the signal paths in the object theory are defined in terms of the *input* and the *output* predicates, so the meta-interpreter has special cases for the meta-interpretation of these predicates. The metatheory predicate *signal_graph* is defined in terms of this meta-interpreter as illustrated below.

signal_graph(T, O, G) ←
object(O, X) & **(5-3)**
usage(O, U) &
solve_construct(T, output(X, U, Out), G)

In the program *signal_graph*, *G* is a signal graph for the observation *O* for the object theory *T* if the following holds: *X* is the name of an object in the object theory that *O* is an observation about, *U* is the name of the usage (the stereo, mode and settings) indicated by *O* for the diagnosis domain, and *G* can be constructed by a metalevel analysis of the object theory predicate *output* and *output(X, U, Out)* is a partial name for the corresponding predicate of the object theory *T*. An example observation *O* could be: *behaviour(s1, [loudspeaker,sp1], cd_player, sound, bad)*.

5.2.3 The Generation of Hypotheses

The set of hypotheses for each type of fault is defined in the metatheory. The hypotheses are generated in relation to an object theory, so if the object theory were to be modified in restricted aspects, for example, in what objects there are, or what their connections are, there would be no need to alter the program for the generation of hypotheses. However, the program would have to be altered if the object theory is replaced with an object theory for another type of diagnosis domain. This is because the domains may differ in what the suitable hypotheses are for the objects in the domain. For example, diagnosis domains that consist of objects connected with some type of directed flow between them are quite different from domains that have a continuous process, such as a blast furnace in which iron is made. Consequently, the set of suitable hypotheses are not the same for different types of domains. In conclusion, as long as we remain in the same class of diagnosis domains when we modify the object theory it is not necessary to change the definition of the set of hypotheses for a type of fault. However, it is not clear how we can distinguish different types of domains, or rather how we can determine when two domains are of the same type. We will discuss this issue further in Chapter 8.

The next step of the diagnosis process after the construction of the signal graph, is to generate the hypotheses for the fault reported by the user. Since the graph tells us which objects take part in the construction of a particular output, only the nodes of the DAG can be responsible for the abnormal observation. Therefore, the only hypotheses that are necessary to construct are those that can be based on the signal graph.

We now present a program for the hypotheses for a fault. It is designed for the stereo system example, but the program would be similar for other diagnosis domains with similar characteristics. The predicate *hypotheses* below is a relation between an object theory, an abnormal observation and a list of hypotheses.

hypotheses(T, O, G, H) ←
    abnormal(O, T) &
    signal_graph(T, O, G) &
    objects(G, Xs) &
    have_settings(T, Xs, SXs) & **(5-4)**
    connections_in_graph(G, Cs) &
    system(O, Sy) &
    mode(O, M) &
    H1= [intern(Xs, Sy, M), settings(SXs, Sy, M), conns(Sy, Cs)] &
    make_individual(H1, H)

This program states that the hypotheses for an observation *O* of an object theory *T* are *H*, determined by a graph *G*, if the following holds: *O* is an abnormal observation for *T*, *G* is the signal graph for *O* in *T*, *Xs* are names for the objects in the signal graph, *SXs* are names for the objects having settings (as determined by *T*), *Cs* are names for the connections between the objects in the graph, *Sy* is the name for the configuration of the diagnosis domain involved in the observation *O*, and *M* is the name for the mode of use of the diagnosis domain. The hypotheses will then consist of the following three groups of hypotheses: (1) internal error in an object in mode *M*, (2) a setting of an object that is incorrect in mode *M*, and (3) an incorrect connection between objects in the configuration *Sy*.

Example:

Let *T* be *stereo*, *Sy* be *s1*, *M* be *cd_player*, and *O* be *behaviour(s1, loudspeaker1, cd_player, sound, bad)*.

The signal graph *G* would then be:

d([loudspeaker1, cable3, amplifier,
d([[cable4, power_supply1], [cable2, CD_player, cable1, power_supply2]])])

Xs would be:

[loudspeaker1, cable3, amplifier, cable4, power_supply1, cable2, cd_player, cable1, power_supply2]

SXs would be:

[amplifier, cd_player]

Cs would be:

[[loudspeaker1, cable3], [cable3, amplifier], [amplifier, cable4], [cable4, power_supply1], [amplifier, cable2], [cable2, cd_player], [cd_player, cable1], [cable1, power_supply2]]

And, finally, *H* would be:

[
intern(loudspeaker1, s1, cd_player), intern(cable3, s1, cd_player), intern(amplifier, s1, cd_player),
intern(cable4, s1, cd_player), intern(power_supply1, s1, cd_player), intern(cable2, s1, cd_player),
intern(cd_player, s1, cd_player), intern(cable1, s1, cd_player), intern(power_supply2, s1, cd_player),
settings(amplifier, s1, cd_player), settings(cd_player, s1, cd_player),
conns(s1, loudspeaker1, cable3), conns(s1, cable3, amplifier), conns(s1, amplifier, cable4),
conns(s1, cable4, power_supply1), conns(s1, amplifier, cable2), conns(s1, cable2, cd_player),
conns(s1, cd_player, cable1), conns(s1, cable1, power_supply2)
]

End of example.

We have now seen how we can take advantage of the object theory, via a metalevel analysis of the theory, when we generate the hypotheses for a particular fault reported by the user. The generated hypotheses may now be taken as a starting point when the diagnosis system tries to locate the fault more precisely. The goal of the diagnosis system is to locate the incorrect property of the object that is responsible for the fault.

Our goals for the representation of diagnosis knowledge were to represent the logical structure (two-level) of knowledge, a modular representation of knowledge and transparency of representation. The two-level structure is clearly retained in the program *hypotheses*, since the object theory is an object of the program—domain and problem solving knowledge are not mixed in a flat representation. As regards the modular representation of knowledge, it is possible to redefine the set of hypotheses by rewriting the program *hypotheses*, without changing the object theory. Conversely, if we were to modify the object theory, we would get a different set of hypotheses since the hypotheses are defined in terms of an object theory. With the signal graph we have a general program for the generation of hypotheses that can be used with different object theories without modification of the metatheory, because the graph is

constructed directly from the object theory. If we add some component to the object theory this component will become part of the signal graph the next time the graph is constructed. Therefore, we can say the program for the generation of hypotheses is modular in the sense that we can regard the object theory as a module that can be modified or replaced. Finally, the program has some degree of transparency since it has a declarative reading as a relationship between an object theory, an abnormal observation and a list of hypotheses.

We have now discussed the first two of the five parts (mentioned at the beginning of the chapter) of the diagnosis process that we regard as central—the handling of observations and the generation of hypotheses. In the following sections we will investigate how we can examine the hypotheses to find the cause of the fault.

5.3 OBJECT LEVEL REFUTATION

A set of hypotheses is not, in general, a satisfactory answer of a diagnosis process, since it only identifies the *possible* causes for a malfunction and a user would not accept to repair every possible cause. A more precise answer can be given if the diagnosis system investigates the hypotheses in some detail. It should be possible to eliminate at least some of the them through an interactive dialogue with the end user. In this interaction the diagnosis system would ask questions to the user regarding the objects of the diagnosis domain. The answers given by the user could indicate that an area of the domain identified by a hypothesis functions correctly, so that hypothesis could be taken out of consideration. In contrast, if the answers indicate that the area malfunctions, then we have found the answer of the diagnosis.

Since the object theory is a model of the diagnosis domain the theory is a prime source of knowledge that can be used in the investigation of the hypotheses. Let us therefore distinguish two methods for investigating hypotheses, based on whether or not they rely on the model of the diagnosis domain. The first uses the model of the diagnosis domain to investigate the hypotheses. This method can be called *object level refutation*. It compares statements of the object theory regarding structure and behaviour of the diagnosis domain with the user's observations of the real world. If the observations confirm the statements, then the hypothesis is refuted—the diagnosis domain is correct with respect to the area identified by the hypothesis. The statements of the object theory must of course be relevant for the hypothesis. For example, the hypothesis that a component has an internal fault could be refuted if all statements about outputs from the component are confirmed by user observations of the actual component. The method of object level refutation will give correct diagnoses provided the object theory is an accurate representation of a non-faulty domain. If the real world observations confirm the theory statements, then the hypothesis cannot identify the cause of the malfunction.

The second method can be called *metalevel refutation* because hypotheses are eliminated on the grounds of knowledge formalized in the metatheory. In its basic form it does not use the model of the diagnosis domain to investigate the hypotheses. The metatheory is based on diagnosis experience of the diagnosis domain, knowledge that is not formalized in the object theory either because it is heuristic or because it would require a more detailed formalization than is expressed in the object theory. The first case illustrates that this method might not always be correct. Therefore, it would be desirable to distinguish between hypotheses that have been eliminated by object level refutation from those that

have been eliminated by metalevel refutation. The rest of this section will describe object level refutation in detail, whereas metalevel refutation will be discussed in section 5.4.

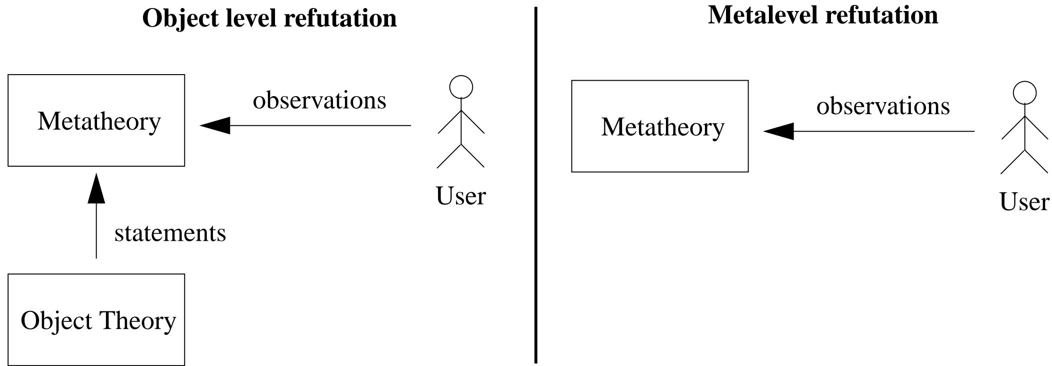


Fig. 5-4: *Object level refutation compares observations from the user with statements of the object theory, whereas metalevel refutation primarily relies on statements of the metatheory.*

Object level refutation tests a hypothesis by deriving statements in the object theory and asking the user if the statements can be observed in the malfunctioning diagnosis domain. We must, however, know what statements to derive for which hypothesis, since not all statements of the object theory are relevant for each. The relation between a hypothesis and the relevant statements can be formalized as a metatheory program, *refutation*, between an object theory *T*, a list *L* of names of statements of *T* and a hypothesis, such that the statements derivable from *T* is a refutation of the hypothesis.

refutation(T, L, intern(X, Sy, M)) ←
 demo(T, output(X, [Sy, M, S], Out)) &
 L = Out
refutation(T, L, settings(X, Sy, M)) ←
 demo(T, standard_settings(X, M, S)) & **(5-5)**
 L = S
refutation(T, L, conns(Sy, X, Y)) ←
 demo(T, configuration_unit(Sy, X1--UV1--Y--UV2--X2)) &
 (X = X1 & L = UV1
 ∨ *X = X2 & L = UV2)*

In the first conjunct, *demo*, of the first clause of this program, *output(X, [Sy, M, S], Out)*, is a partial name (containing metavariables) for the object theory formula

$$\exists X \exists Sy \exists M \exists S \exists Out(Output(X,[Sy,M,S],Out)).$$

Let us now investigate the use of *refutation* for a particular hypothesis. Assume that we try to refute a hypothesis about an internal fault in a loudspeaker, so that we have the goal

refutation(stereo, L, intern(loudspeaker1, s1, cd_player))

which would result in the following call to *demo*

demo(stereo, output(loudspeaker1, [s1,cd_player,S], Out))

where

output(loudspeaker1, [s1,cd_player,S], Out)

is a partial name of the object theory formula

`∃S ∃Out(output(loudspeaker1,[s1,cd_player,S], Out) ).`

The result of the *refutation* query is that *L* will be a list of names of statements derived from the object theory predicate *output*.

L = [signal(s1, loudspeaker1, cd_player, volume, 79 dB) ,
behaviour(s1, loudspeaker1, cd_player, sound, good)]

We assume in the object theory certain standard settings if none is specified in the query to the theory, for instance, the volume control has a standard setting that allows us to derive the value of 79 dB above. This example illustrates how we relate a hypothesis to object theory statements. The hypothesis *intern(X,Sy,M)* points out the problem area "internal function of an object *X* of a system *S*y in mode *M*" and the object theory definition of *output* formalizes the output signals and observable behaviours of the object *X*, which depend on the internal function of the object. Therefore, it can be used for deriving statements from the object theory to test the hypothesis.

The last step of the refutation method is to verify if the statements of the object theory can be observed in the user's malfunctioning diagnosis domain. If the statements are confirmed by the user, then the hypothesis is refutable, i.e. it does not identify the fault and can thus be removed from consideration. The method of object level refutation can be represented in the metatheory as the following program.

object_refute(T, Obs, Hyp) ←
refutation(T, L, Hyp) & **(5-6)**
observable(L, Obs)

The program *object_refute* is a relation between a theory *T*, a list of user observations *Obs*, and a hypothesis *Hyp*. The hypothesis is refutable when: (1) there is a list *L* of names of statements derivable in the object theory *T*, and (2) the user answers *Obs* determine that the statements of *L* can be observed by the user.

What criteria can be applied in deciding whether the statements have been observed by the user and the hypothesis regarded as refuted? There are two aspects of this question: First, how many of the statements should be required for a refutation? The second aspect concerns the degree of verification by the user, that is to say how certain the user is of the observation and what interval of values can be accepted for an observation to count as positive.

The obvious answer would be to require the user to observe every statement in the list, exactly to the value of the statement, being 100 percent certain of the observations. This would mean that if there is at least one statement in the list that cannot be definitely confirmed by user observations, then the hypothesis points out a fault. However, these requirements could be relaxed to various degrees, most likely differently in different domains. In some domains, such as the domain of the stereo system one would require that all the statements be observable by the user, quite close to the value specified. We find it reasonable to assume that in other domains, such as in a domain of some biological or chemical

system one would probably not require the user to be able to observe every statement and values would be allowed to vary within a greater interval, since biological and chemical systems vary more than electronic systems. In our discussion about normal and abnormal observations in section 5.1, we mentioned a predicate *equal* that would formalize for a particular domain when two values are equal. This predicate would naturally be used for determining whether the user's observation confirms the statement of the object theory. Furthermore, the user's degree of certainty of observations could be taken into account by defining appropriate metatheory predicates for certainty handling, although it is far from clear how they would be defined. One of the factors that would determine these choices is the precision of the model that is formalized in the object theory.

We have discussed an approach for investigating hypotheses in a metatheory using a model of a diagnosis domain by relating hypotheses with (names of) formulas in a theory of a diagnosis domain. The relation is given in a metatheory relation *refutation* and is illustrated with three types of hypotheses. The approach uses a modular representation of diagnosis knowledge, in the sense that the object theory can be seen as a module that may be modified without changing the metatheory, as well as the other way round.

5.4 METALEVEL REFUTATION

In this section we will discuss the second method for the refutation of hypotheses. This method uses primarily knowledge that is not present in the object theory which represents the diagnosis domain, but rather knowledge based on experience. Knowledge acquired from problem solving in a particular domain made the first generation expert systems powerful problem solvers in some domains, where it was possible to collect enough quality knowledge to achieve expert performance. Knowledge was usually represented in a form compiled from knowledge of various kinds, for example, structural, causal, functional, associational and experiential knowledge, into a compact form for a particular domain. This knowledge expressed associations between symptoms and conclusions that could be very effective for problem solving. As discussed previously in this thesis, the use of compiled knowledge is not without problems. One of them is to know what knowledge to modify if the structure or function of the domain changes, since the knowledge is based on the configuration of a particular domain and it is not always easy to trace such knowledge back to the structure of the domain. Both the acquisition and modification of knowledge thus requires experts in the domain, which makes the development and maintenance of these systems expensive. Therefore, model-based systems with an explicit representation of a model for a domain (e.g. a structural and functional model) have been developed as an alternative to the first generation expert systems. However, since compiled knowledge can be useful in diagnosis systems because of its compactness and effectiveness, it would be valuable to integrate the use of such knowledge with the use of model-based knowledge. Below we will discuss an approach for integration.

5.4.1 Two Kinds of Compiled Knowledge

We will distinguish two kinds of compiled knowledge that we would like to integrate with the use of model-based knowledge. The first kind is compiled knowledge of a heuristic character, i.e., problem solving knowledge that is often true but not always. An example of heuristic knowledge is the following rule: "If a loudspeaker of the stereo system does not work and the settings of the amplifier have been checked, then the connections between the components are faulty." This rule is clearly not always true, but it is a useful heuristic for finding a diagnosis. It compiles knowledge of structural, functional and experiential character in a compact form. Knowledge about the connections of the stereo

system (structural knowledge), about the functionality of the loudspeakers, and experiential knowledge from diagnosing stereo systems is here compiled into a single rule. We do not include heuristics in the object theory because they constitute problem solving knowledge which we do not want to mix with knowledge about the function and structure of the domain. However, it would certainly be convenient to use domain specific heuristics together with the object theory because they can be very powerful for problem solving. Therefore, we propose to include heuristics in the metatheory as a guidance for the use of the model-based knowledge in the object theory.

The second kind of compiled knowledge has a complexity that surpasses the level of abstraction of the domain theory. Such knowledge is inconvenient to represent explicitly in the object theory. The reason is that an explicit representation would force the level of abstraction lower than that chosen by the designer. In the design of a domain theory one must choose a level of detail that is not too difficult and expensive to represent and has an acceptable computational complexity. This issue is an instance of the general problem of granularity level of representation as discussed by Genesereth and Nilsson [23]. The granularity of a representation determines what can be deduced from the represented knowledge—a fine granularity implies that we can deduce very detailed information, whereas a coarse granularity implies less detailed information which, however, may be easier to reason with. In any representation of knowledge we must determine what granularity level of representation we need—there is no generally applicable level that is always the best, in contrast, we must choose a level depending on our needs which of course vary from system to system. Also Console and Torasso [12] discuss the problem of constructing a complete model. They argue that any model will contain abstractions, therefore it is not complete in the sense that it does not formalize the domain completely.

An example of this choice is the following complex compiled knowledge in the stereo system: "If a loudspeaker of the stereo system produces some sound, then the power-supply cannot be faulty." The knowledge is always correct because we know that a loudspeaker requires that at least some component has electricity for a loudspeaker to function (we disregard the possibility of a reduced mains power in our example). The mechanism of how components depend on the power supply in the stereo system, which is behind this knowledge, is not represented in our object theory. It would be possible to represent it as model-based knowledge in the theory for the diagnosis domain, but that would increase the complexity of the theory further than that chosen in our example. If we represented the power-supply knowledge in the object theory, we would not be able to deduce enough new interesting things for the diagnosis to outweigh the increased complexity of the representation.

Another reason for putting knowledge in the metatheory can be that the mechanisms behind the compiled knowledge are difficult to understand for the expert or not completely understood. A third reason for representing complex knowledge in the metatheory can be computational. A rule could be derivable in the object theory, but also represented in the metatheory as a derived rule because of the computational cost for its derivation.

Both kinds of compiled knowledge—heuristic and complex—can be represented in the metatheory of the diagnosis system. The first kind can be represented as a guidance for the selection of hypotheses to refute. Since such knowledge is not always correct it should not be allowed to rule out hypotheses, but it can serve as a guide for ordering hypotheses. The second kind, however, can be allowed to rule out hypotheses since it should always be correct in the domain. We discuss the representation of heuristic compiled knowledge in section 5.4.3 and the representation of complex compiled knowledge in the section below.

5.4.2 A Representation of Complex Compiled Knowledge

We aim to integrate the use of complex compiled knowledge with the use of model-based knowledge through the methods of refutation. Model-based knowledge is used in object level refutation to refute hypotheses, whereas in metalevel refutation compiled knowledge can be used to refute hypotheses. Both methods would discuss the same kind of objects (hypotheses), but base their reasoning on different kinds of knowledge, which gives us an integration of compiled and model-based knowledge in the refutation process. What differs between the methods are the grounds for a refutation. Object level refutation bases the refutation on names for object theory formulas which guarantee that the conclusions are correct because the formulas are expressed in terms of the structure and function of the domain and derived in the object theory. Metalevel refutation bases the refutation on compiled knowledge so we can go directly from observations to conclusions without reference to the object theory. This makes the method (potentially) powerful, because there is no need to formalize in the object theory the details of the relations between observations and conclusions, but for the same reason the method is sensitive to modifications in the domain. Again, this is a factor that can influence the choice of granularity level of representation. If the domain of the system is frequently modified it may be better to transform some of the complex compiled knowledge to model-based knowledge in the object theory, since it is usually easier to trace changes in the domain to the model of the domain (in the object theory) than to compiled knowledge (in the metatheory).

Let us discuss three examples of a possible representation of compiled knowledge suitable for metalevel refutation. In the first example we will see how we can represent compiled knowledge as a relation between the observations *Obs* of the user and a hypothesis. An example from section 5.4.1 will illustrate this representation. The rule "If a loudspeaker of the stereo system produces some sound, then the power-supply cannot be faulty", would be represented as the following clause.

meta_refute(Obs, intern([power, X], Sy, M)) ←
  observable(O, Obs) & **(5-7)**
  O = behaviour(Sy, [loudspeaker, X1], M2, volume, Z dB) &
  Z > 0

This rule means that all hypotheses of the form *intern([power, X], M)* are refutable when (1) there is an observation

behaviour(Sy, [loudspeaker, X1], M2, volume, Z dB)

which says the user has observed that there is a volume of *Z* dB from a loudspeaker, and (2) *Z* is higher than *0*. When the above clause is true all hypotheses concerning internal faults in the power supply are refutable, so if the system has two hypotheses about internal faults in the power supply, this rule would refute both hypotheses. This is a representation of compiled knowledge that is expressed entirely in terms of metalevel knowledge—no reference is made to any explicit representation of the structure or function of the domain. If the domain is modified, this rule may also need modification.

Below we see the general form for clauses in this relation.

meta_refute(Obs, Hyp) ←
  observable(O, Obs).

The above relation represents that a hypothesis *Hyp* is refuted when an observation *O* can be observed by the user as indicated by the observations *Obs*.

With a metalogic approach to the representation of diagnosis knowledge it is possible to represent generalizations of compiled knowledge. Our approach is to generalize compiled knowledge by stating it relative an object theory so that the knowledge is general over some class of object theories for different systems. Furthermore, by generalizing compiled knowledge, our metatheory can be kept more compact since one general rule can replace more numerous specific rules. As the second example let us therefore extend the relation *meta_refute* to include an object theory, so that we can represent a general rule regarding hypotheses about the power supply. A rule such as "If an object of the stereo system produces some behaviour or signal that is *normal* for the stereo system, then the power-supply cannot be faulty", could then be represented as the following clause.

meta_refute(T, Obs, intern([power, X], Sy, M)) ←
observable(O, Obs) & O = Otype(Sy, X2, M2, Ty, V) & **(5-8)**
member(Otype, [signal, behaviour]) &
normal(O, T)

This clause states that any observation of a signal or a behaviour that is normal in a theory *T*, implies that all hypotheses regarding internal faults in the power supply can be refuted. The idea behind this compiled knowledge is that any signal or behaviour in the stereo system domain requires electricity, so therefore it is possible to rule out the power supply as the problem. We have generalized the first clause by making the second clause relative an object theory, so if the object theory is modified (e.g., new components) the second clause would not have to be modified. Thus the second clause is less sensitive than the first to changes in the domain, but it is still a powerful compilation of knowledge. Although the clause is general it still contains compiled knowledge since there is no explicit representation of how the power supply affects the function of components—this is still implicit, compiled knowledge. Furthermore, this general rule replaces specific rules for each kind of normal output of the stereo system. Thus the metalogic approach helps us to keep our representation compact.

To extend the second example, we can draw further conclusions in the stereo domain from the fact that there is a is normal observation. Since a component with a normal signal or behaviour requires electricity, we can draw the conclusion that the cable between that component and the power supply, as well as the connections between the component and the cable, are OK. This can be represented as below, where the refutable hypotheses are represented as a list.

meta_refute(T, Obs, [intern([power, X], Sy, M),
(conns(Sy, Y1,[power,X]), conns(Sy,X2,Y1), intern(Y1,Sy, M2))]) ←
observable(O, Obs) &
O = Otype(Sy, X2, M2, Ty, V) & **(5-9)**
member(Otype, [signal, behaviour]) &
normal(O, T)

When this clause is true all hypotheses *intern([power, X], Sy, M)* and all combinations of the hypotheses *conns(Sy,Y1,[power,X])*, *conns(Sy,X2,Y1)* and *intern(Y1,Sy, M2)* are refutable. This concludes our second example.

A third, more advanced, example of how complex compiled knowledge can be formalized as metalevel reasoning will discuss different object theories. This would make it possible to write programs that reason hypothetically over distinct object theories. If we have a relation *modify_theory* between a theory *T1*, a set *Obs* of observations and a theory *T2* where *T2* is the result of modifying *T1* such that the observations in *Obs* are statements of *T2*, we can reason with different object theories. An example would be causal reasoning about which things affect what in a domain. For instance, if two observations *O1* and *O2* are abnormal for a theory *T1*, but *O2* is normal for *T1* modified for *O1*, then we could draw the conclusion that *O2* is a result of *O1* (i.e., *O1* explains *O2*), and therefore exclude the possibility of an internal fault in the component that *O2* mentions. That is, *O2* is a consequence of *O1* and not a proper fault. We could represent this reasoning as the following two programs.

```
explains(O1, O2, T) ←
        abnormal(O1, T) &
        O1 = signal(Sy,X,M,Ty,V) &
        abnormal(O2, T) &                                   (5-10)
        modify_theory(T, [out_signal(O1,I)], T2) &
        normal(O2, T2)
```

This program means that an observation *O1* can explain an observation *O2* for a theory *T* when both observations *O1* and *O2* are abnormal for *T*, but *O2* is normal for a theory *T2* of which *O1* is an assertion. The second program for this example is *meta_refute*:

```
meta_refute(T, Obs, intern(X2, Sy, M) ) ←
        observable(O1, Obs) &
        observable(O2, Obs) & O1 ≠ O2 &                     (5-11)
        explains(O1, O2, T) &
        object(X2, O2)
```

Here we let a clause of the *meta_refute* relation represent the above example. The clause states that all hypotheses of the form *intern(X2, M)* can be refuted when we have two different observations *O1* and *O2*, where *O1* explains *O2,* and *X2* is the object that *O2* mentions.

We can modify program (5-11) to be more effective by introducing the signal graph into *meta_refute*. Recall that the graph describes the signal paths for the initial abnormal observation reported by the user and the direction of the signals in the stereo system. The root of the signal graph is thus the object that displays the abnormality observed by user. We can take advantage of this to determine which hypotheses can be ruled out as consequences of the real fault. This program states that the case where there are two observations *O1* and *O2*, *O1* is a signal of an object *X1* and *O1* explains *O2* (*O2* would be normal for a theory modified for *O1*), implies that *O1* causes *O2*, thus all objects following *X1* in the signal graph are actually OK and in consequence the *intern* hypotheses about these objects are refutable.

```
meta_refute(T, Obs, G,  H) ←
        observable(O1, Obs) &
        O1 = signal(Sy, X1, M, Ty, V) &
        root(X2, G) &
```

object(X2, O2) & **(5-12)**
observable(O2, Obs) & O1 ≠ O2 &
explains(O1, O2, T) &
objects_after(X, X1, G) &
make_individual([intern(X, Sy, M)] , H)

Note the condition where observation *O2* mentions the root of the signal graph which ensures that the rule refutes as many hypotheses as possible.

In the above program we see that metalevel reasoning where an object theory is regarded as a term in the language gives us an approach for writing declarative and powerful reasoning programs. Moreover, we find in the above program a modular representation. It is modular since the program is general over different theories, so it works even if we replace the theory *T* for another theory *T'* with a different set of stereo components or components of some similar technical system. So, despite the fact that we represent complicated compiled knowledge we have the means to preserve a certain generality of the knowledge.

Our motivation for giving the preceding examples has been to illustrate our approach to represent compiled knowledge separately from model-based knowledge, without giving up the ability to use them together. It has not been our intention to give rules that are general for any class of diagnosis domains, only to exemplify compiled knowledge for a particular diagnosis domain. However, it is possible to distinguish some generalities of the metatheory. Some of the clauses for *meta_refute* may have to be modified for a different stereo system. In contrast, the clauses for *object_refute* would not have to be modified. We can thus distinguish two parts Γ and Φ of the metatheory where Γ is the general diagnosis knowledge in the theory such as *object_refute* whereas Φ is the compiled knowledge which is more specific for the domain. We will now repeat Fig. 5-4 with some modifications to illustrate this structure of the metatheory in relation to object level and metalevel refutation. Below we find this as Fig. 5-5.

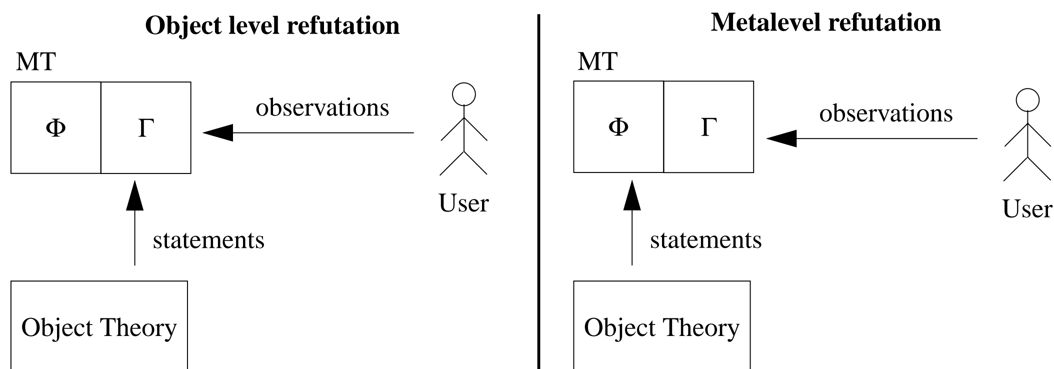


Fig. 5-5*: Object level refutation compares observations from the user with statements derived from the object theory via general diagnosis knowledge* Γ*, whereas metalevel refutation primarily relies on compiled knowledge* Φ *in the metatheory, possibly assisted by model-based knowledge in the object theory.*

We regard compiled knowledge as quite domain specific. We have, however, provided examples of some generality in compiled knowledge, as discussed above. With our approach for the representation of compiled knowledge there are prospects for generality also in compiled knowledge—

not only in model-based knowledge. Consider another diagnosis domain than our example, with another object theory and another set of clauses for *meta_refute*. If that domain were to be modified, some of the compiled knowledge would have to be changed as well. The extent of modifications of that compiled knowledge would depend on how generally the compiled knowledge had been formulated.

We have illustrated how complex compiled knowledge for the domain of our stereo system can be represented in a metatheory. The generality of the general metalevel knowledge Γ goes over different stereo systems—it does not apply to any larger class of diagnosis domains without modifications. However, there seems to be a potential to generalize the metatheory for a class of domains. We will discuss these possibilities further in Chapter 8.

In conclusion, we find that metalevel refutation of hypotheses with theories and observations provides us with a way of integrating the use of compiled knowledge with the model-based knowledge used in object level refutation. Both kinds of knowledge can be integrated in the refutation process via the two methods. We have also seen that metalevel refutation provides a further integration of knowledge if we can write clauses of compiled knowledge, general over different theories. In our view, the integration does not come at the cost of losing the logical structure, modularity or transparency of the representation, which were our goals for the representation of diagnosis knowledge. The examples given above illustrate some degree of modularity and transparency, since the object theory and the clauses of the compiled knowledge can be regarded as modules and the compiled knowledge is formulated as declarative sentences. The logical structure of the knowledge is clearly retained when generalized and compiled.

5.4.3 A Representation of Heuristic Compiled Knowledge

In this section we discuss how heuristic compiled knowledge can be represented with our approach and indicate how heuristics can be used as a guidance in the diagnosis process. We attempt to show that heuristic knowledge can be given a representation that meets our goals for the representation of diagnosis knowledge—to preserve the logical structure of diagnosis knowledge and to provide a modular representation that is also transparent for the knowledge engineer.

As previously discussed, heuristic compiled knowledge should not be allowed to give us conclusions—it should only guide the diagnosis process. The methods of object refutation and meta refutation give conclusions regarding the hypotheses of the diagnosis and this is correct since they refer to knowledge that is always correct (in the domain). For meta refutation, complex compiled knowledge is represented as a set of clauses in the relation *meta_refute*. We would like to represent heuristic compiled knowledge in a similar fashion because the kind of knowledge to be expressed is rather similar to complex compiled knowledge, the difference being that heuristics can only be taken as recommendations. We can represent heuristic compiled knowledge as a separate relation, *probably_refutable*.

The relation *probably_refutable* relates a hypothesis *H*, a theory *T*, the user observations *Obs* and the hypotheses *THs* that have been tested in the diagnosis. The relation is true when the hypothesis *H* can be recommended for refutation. Recall the example in section 5.4.1 of heuristic knowledge: "If a loudspeaker of the stereo system does not work and the settings of the amplifier have been checked, then the connections between the components are faulty." Here, the knowledge clearly refers to the progress of the diagnosis process. By including the tested and refutable hypotheses *THs* in the relation *probably_refutable* we can refer to the progress of the diagnosis. This example can be represented as a clause of the relation *probably_refutable* as illustrated below.

probably_refutable(T, Obs, THs, [(conns(Sy, X2, Y1), conns(Sy,Y1,X1))]) ←
observable(O, Obs) &
object(X2, O) &
type(X2, loudspeaker) & **(5-13)**
abnormal(O, T) &
type(X1, amplifier) &
member(settings(X1,Sy,M), THs)

This clause states that the hypotheses *conns(Sy, X2, Y1)* and *conns(Sy,Y1,X1)* can be recommended for (object) refutation when there is an observation *O* in the user observations *Obs*, and the object of that observation is a loudspeaker, and the observation *O* is abnormal for the theory *T*, and the hypothesis regarding the settings of the amplifier is tested (refuted).

With a set of clauses such as the one above we can get a recommendation of which hypotheses to test. Each hypothesis in the list of hypotheses generated by the program *hypotheses* (see section 5.2.3) could be tested for the property of *probably_refutable* as an indication of its potential for refutation. The hypotheses that are *probably_refutable* would be tried first as they have good potential. If we would like to order the hypotheses recommended by the heuristics it would be possible to extend the relation *probably_refutable* with an argument for a measure for ordering of hypotheses. For instance, the clause above could be given a measure of 0.9 if we were rather certain that the heuristic is often correct, whereas another heuristic might be given a lower measure. If the probabilities for component failure were known these could also be used. These measures would then be a ground for ordering the hypotheses for testing. The design of different measures for certainty could of course be further elaborated, for example as in GDE (see section 2.2) but we only indicate a way to incorporate them in the representation.

The order to test the hypotheses may, however, depend on other factors than those represented in the heuristic compiled knowledge. Since the testing of hypotheses often involves asking questions to the user of the diagnosis system, user interaction factors are of great importance. Such factors include the difference between questions easy to answer and questions whose answers may require a substantial amount of work from the user, as well as considerations of the naturalness of question sequence when a particular sequence of questions would be preferred by the user. A basic sorting order for hypotheses could therefore be imposed that sorts hypotheses according to how easy it is for the user to observe the statements required for the refutation of a hypothesis. Another factor is the potential effect of a question regarding the number of hypotheses that could be refuted by a certain question to the user. We will briefly touch upon these questions in the following section on the diagnosis strategy.

5.5 A REPRESENTATION OF A DIAGNOSIS STRATEGY

In the present chapter we have discussed the first four of the five parts of the diagnosis process that we regard as central and want to represent in the metatheory. The five parts are: the handling of observations, the generation of hypotheses, the refutation of hypotheses, the ordering of the hypotheses to refute and handling of the user interaction. In this section we will discuss how these five parts can be integrated as a diagnosis strategy.

We shall discuss a definition of diagnosis as a ternary relation, look at an example strategy for establishing a diagnosis and discuss the representation of the strategy as a logic program in the metatheory. We will have the definition of diagnosis as a guide for the design of the strategy.

5.5.1 A Definition of Diagnosis

Our purpose with a definition of diagnosis is to get a declarative characterization of it. An argument for characterizing diagnosis as a ternary relation is that there are three entities involved—a domain, a set of user observations and a fault in the domain. To get a logical characterization of diagnosis we should include these three entities in the characterization—thus a relation, rather than a characterization as a fault, an unary relation, is suitable. In the metatheory a fault is identified by a hypothesis *H*, the user observations are collected in a list *Obs* and the domain is formalized as an object theory *T*. We can thus define diagnosis as a relation in the metatheory between an object theory *T*, a set of user observations *Obs* and a hypothesis *H*. The relationship between these objects is that *Obs* contains an observation, abnormal for *T* and related to an area of the domain identified by *H*. Informally, *H* is a diagnosis for the abnormal observations in *Obs*, in the domain formalized in *T*. Below we find the definition of *diagnosis*.

∀T∀Obs∀H(diagnosis(T, Obs, H) ↔
theory(T) & observations(Obs) &
∃O∃G∃Hs(member(O, Obs) & abnormal(O, T) &
hypotheses(T, O, G, Hs) & member(H, Hs) &
¬ meta_refute(T, Obs, G, H) &
¬ object_refute(T, Obs, H)))

The relation *diagnosis* is true when we have an object theory *T*, a set of user observations *Obs*, and a hypothesis *H*, for which the following holds: There is an observation *O* in *Obs* that is abnormal with respect to *T*, the result of generating hypotheses from *T*, *O* and the signal graph *G* are *Hs*, and *H* is a hypothesis in *Hs* which is neither refutable in the metatheory, nor with the object theory *T*. Recall that we have discussed these relations previously in this chapter. *abnormal* was discussed as program (5-2), *hypotheses* as program (5-4), *object_refute* as program (5-6), and, finally, *meta_refute* as programs (5-7) to (5-12).

We now take the definition of *diagnosis* above as a starting point for the implementation of a strategy. There is not only one unique strategy that can find a diagnosis as defined in *diagnosis*. However, we look at one example strategy that meets this definition.

5.5.2 A Diagnosis Strategy

The basic idea for this strategy is to ask the user for an object theory and an observation of some faulty behaviour (or signal), e.g. that a loudspeaker does not produce any sound and then generate the hypotheses. The strategy then tries to refute (eliminate) each hypothesis until one is found that cannot be. This hypothesis will then be presented as the answer of the diagnosis, together with the user's observations and the object theory. To explain in more detail; after generating the hypotheses the strategy proceeds as follows. The relation *probably_refutable* is used to sort the hypotheses in the order of likelihood, as discussed in section 5.4.3. A simple selection criterion is then used to pick hypotheses for testing—e.g. the first hypothesis. The refutation process then first tries meta refutation, because it might use some compiled knowledge that gives us a short-cut in the diagnosis. If meta refutation does not eliminate the hypothesis, object refutation will be tried. If *object_refute* is true for the hypothesis it is moved from the list of active to the list of tested hypotheses *THs* and the rest are tried. On the other hand, if *object_refute* is not true, the hypothesis and the user's observations are presented as a diagnosis

for the object theory, because the it identifies an area of the domain that does not show the same behaviour as asserted by the object theory. We give an algorithm for this strategy.

Diagnosis Algorithm

1. Let the user select an object theory *T*.
2. Let the user give an observation *O* that is abnormal in *T*.
3. Generate hypotheses *Hs* for faults based on *O* and *T*.
4. Sort all *H* in *Hs* that are *probably_refutable* before those that are not.
5. If there is a hypothesis *H* in *Hs*,

   then select the first *H* in *Hs*,

   else stop (the system cannot determine a diagnosis).
6. If *H* can be refuted with *meta_refute*,

   then move *H* from *Hs* to the list of tested hypotheses *THs* and go back to 4,[1]

   else if *H* can be refuted with *object_refute*,

   then move *H* from *Hs* to *THs* and go back to 4, [2]

   else go to 7.
7. Present the hypothesis *H* and the observations for the user and stop.

In this algorithm we have no specification of how and when questions of observations should be put to the user. At the end of section 5.4.3 we indicated this problem. A simple solution to the problem would be to ask all that are prompted by *meta_refut*e and *object_refute*, as well as by *probably_refutable*, i.e. when an observation is required in e.g. *meta_refute* a question would always be asked about the observation, as long as it had not been asked before. This is essentially the "Query-the-User"-approach of Sergot [43]. To avoid a potentially unfocused questioning of the user if we have many *meta_refute* clauses, an alternative solution would be to ask only those prompted by *object_refute* and let the answers be the only way to get information from the user.

A third, more advanced, solution would be to ask those prompted by *object_refute*, but also investigate the potential of the clauses of *meta_refute* to asses the usefulness of questions put to the user. Such investigation could estimate the number of hypotheses that could be ruled out by a positive answer to a particular question, by comparing the list of hypotheses with the hypotheses in each clause. The utility of questions for *probably_refutable* could be handled similarly. A simple approach to this solution is to separately define a question to the user, *potential_question*, for each clause of *meta_refute*. It will prompt the user for an observation that could make the clause true. The only formal connection between the *meta_refute* clause and the question would be that the same hypotheses are given in the clause as in the definition of the question. However, informally the system designer would know that the question is connected to the clause. Since all clauses of *meta_refute* can be tested each time a hypothesis is attempted for meta refutation there would be no need for any further formal connection. The system can use *potential_question* to evaluate how many hypotheses might be refuted if a positive answer is given by the user to the question. For example, consider program (5-9), repeated below:

---

1. The user may have reported new observations when the *meta_refute* clauses have been tried, which may affect the sorting of the remaining hypotheses.
2. The testing of the *object_refute* clauses may also have generated new user observations.

meta_refute(T, Obs, [intern([power, X], Sy, M),
(conns(Sy, Y1,[power,X]), conns(Sy,X2,Y1), intern(Y1,Sy, M2))]) ←
observable(O, Obs) &
O = Otype(Sy, X2, M2, Ty, V) &
member(Otype, [signal, behaviour]) &
normal(O, T)

A question can be defined for this clause, “Please indicate some normal activity in the stereo”, in *potential_question*. Note that the list of hypotheses is the same as in the clause.

potential_question(‘Please indicate some normal activity in the stereo’,
Hs, THs, [intern([power, X], Sy, M), **(5-14)**
(conns(Sy, Y1,[power,X]), conns(Sy,X2,Y1), intern(Y1,Sy, M2))]))

With this simple approach we can also put conditions on the questions by adding them to the *potential_question* clauses. For instance, program (5-12) repeated below, about a fault that is caused by another fault can have a question with a condition (5-15).

meta_refute(T, Obs, G, H) ←
observable(O1, Obs) &
O1 = signal(Sy, X1, M, Ty, V) &
root(X2, G) &
object(X2, O2) &
observable(O2, Obs) & O1 ≠ O2 &
explains(O1, O2, T) &
objects_after(X, X1, G) &
make_individual([intern(X, Sy, M)] , H)

The question for this clause would be “Please indicate some abnormal signal in the stereo system”. We choose to state that this question should only be asked when there are only hypotheses about internal faults left to investigate. The reason being that we do not want the diagnosis system to ask for signals (*intern* hypotheses are refuted via user observation of signals), which are somewhat inconvenient for the user to measure, as long as there are other hypotheses to investigate. Therefore, this question has as a condition that there should only be *intern* hypotheses left to investigate. *Hs* contains the hypotheses that remain to be checked.

potential_question(‘Please indicate some abnormal signal in the stereo’,
Hs, THs, [intern(X, Sy, M)]) ← **(5-15)**
¬ (member(Hyp(_), Hs) & Hyp ≠ intern)

To incorporate this solution to the problem of putting questions to the user in our diagnosis algorithm we would add an extra step “4a. Evaluate questions” to the algorithm.

The approach of connecting questions to *meta_refute* clauses is limited since if we have many *meta_refute* clauses in the diagnosis system, it might be difficult for the programmer to know which

question relates to which clause. Potentially, the ‘wrong’ question could be asked if several clauses of *potential_question* have the same hypotheses. However, all potential questions should be evaluated and only the one with the best potential for refuting hypotheses should be asked. The approach needs further study to avoid these problems.

A final problem for the design of the diagnosis strategy also concerns user interaction. The user may at some point of the diagnosis process feel that a particular question is difficult to answer or put at the wrong time. For example, a question regarding the internal status of a stereo component could be difficult to answer because it requires the user to dis-assemble the component. A solution to this problem could be to let a user interface management system (UIMS, see [28] for an overview) decide whether a question should be asked or not or whether another hypothesis should be tested first instead. To use an UIMS, we would insert a step for the UIMS in the diagnosis algorithm after step 5. It is, however, not trivial to decide which considerations belong to the user interface management and which to the diagnosis strategy. The selection of which hypothesis to test would probably belong to the strategy but (as now discussed) this is not entirely clear. We will not investigate the consequences of using a separate UIMS in depth in this thesis, but in Chapter 7 we will discuss related aspects of the user interface.

5.5.3 A Representation of the Diagnosis Strategy as a Logic Program

Finally, in this section we will discuss a representation of the example diagnosis strategy as a program in a logic programming language. The diagnosis algorithm presented above can be implemented in different programming languages, which may be suitable for different purposes, but considering that we have a definition of *Diagnosis* in predicate logic, it is rather straight-forward to provide a representation as a logic program. The particular problems we face in representing the algorithm as a logic program relates to our need to retain the answers given by the user when asked about observations in the domain. In general, when an algorithm is represented as a logic program, more or less extensive use is made of the backtracking mechanism of logic programming languages. For example, a generate-and-test technique is often used where an object is generated and then tested for some condition. If it is not true for the object, backtracking occurs and the next object is generated (and tested). In the diagnosis algorithm, we have this situation for the hypotheses where a hypothesis is selected and then tested with *meta_refute* and *object_refute*. It would be natural to use backtracking to implement this, if it was not the case that we want to keep the answers given by the user. An alternative implementation is to use a recursive program that does not backtrack. With this technique, we can represent the diagnosis algorithm as a recursive program that brings along the answers given by the user in the recursive calls of the program. We also represent the relations *meta_refute* and *object_refute* as logic programs *object_refutation* and *meta_refutation* each with an extra argument *R1* and *R*2 to indicate the result of the refutation (success or failure). This way we do not loose the user answers on backtracking since the programs never fails. We represent the algorithm as two logic programs *diagnosis_strategy* and *diagnosis*. In this representation of the diagnosis algorithm, we assume that the logic programming language uses a computation rule left-to-right, i.e. the atoms selected for execution are selected from left and the clauses are selected from the top first. This is the case for, e.g., standard Prolog.

In this logic program we have not specified how the hypotheses should be sorted or which hypothesis to select. It would be possible to use *probably_refutable* in *sort_hypotheses*—nothing in the program prevents this. The method of putting questions to the user is not specified, but again, nothing in the program prevents the use of any of the three alternatives discussed above. However, we have included a call to *evaluate_questions* to check the *potential_question* predicate, possibly ask a question

to the user and try the *meta_refutation* program for the hypotheses that can be potentially refuted. Furthermore, a User Interface Management System could be used in *selected_hypothesis* to control the selection of which hypothesis to test.

```
diagnosis_strategy ←
        select_theory(T) &
        select_observation(Obs) &
        member(O, Obs) &
        abnormal(O,T) &
        hypotheses(T, O, G, Hs) &                                   (5-16)
        diagnosis(T, Obs/Obs2, G, Hs, []/THs, H) &
        present_diagnosis(T, Obs2, H)

diagnosis(T, Obs/Obs, G, Hs, THs/THs, H) ←
        empty(Hs) & empty(H)
diagnosis(T, Obs/ObsN, G, Hs, THs/THsN, H) ←
        evaluate_questions(T, Obs/Obs1, G, Hs, THs/THs1) &
        sort_hypotheses(T, Obs1/Obs2, Hs, Hs2) &
        selected_hypothesis(H1, Hs2) &
        meta_refutation(T, Obs2/Obs3, G, H1, R1) &
        (       R1=refutable &
                append([H1], THs1, THs2) & remove(H1, Hs2, Hs3) &
                diagnosis(T, Obs3/ObsN, G, Hs3, THs2/THsN, H)
        ∨                                                           (5-17)
                R1=unrefutable &
                object_refutation(T, Obs3/Obs4, H1, R2) &
                (       R2=refutable &
                        append([H1], THs, THs2) & remove(H1, Hs2, Hs3) &
                        diagnosis(T, Obs4/ObsN, G, Hs3, THs2/THsN, H)
                ∨
                        R2=unrefutable &
                        ObsN = Obs4 &
                        H = H1
                )
        )
```

5.5.4 Diagnosis of Multiple Dependent Faults

In the previous discussion we have assumed that a single fault is responsible for the first observed abnormality. One reason for this assumption has been to keep the presentation simple. There is also the reason of computational complexity, since the assumption of multiple faults can lead to an exponential number of hypotheses with regard to the size of the diagnosis object. This is due to the number of combinations of hypotheses. For example, a diagnosis object with five components could give rise to 206 hypotheses $(206 = 1 + (5 \times 4 \times 3 \times 2) + (5 \times 4 \times 3) + (5 \times 4) + 5)$ if any combination of

components could be faulty (limiting the hypotheses to components). A device with 10 components could result in 6 235 301 hypotheses and one with 20 could give up to 4 180 411 311 071 440 001.

In [19] methods to limit this complexity are introduced, as discussed in Chapter 2. These methods could be incorporated into our diagnosis system. Multiple faults could be handled analogously to GDE in our system. Our metalevel analysis of the object theory to construct the signal graph is analogous to GDE's propagation of values through the device models to predict the output values of the device and in this way identify conflicts. In GDE minimal candidates are generated from a conflict by taking one component from the conflict to construct each minimal candidate. When a new conflict that is not explained by any candidate, new minimal candidates are generated by examining the immediate supersets of the previous candidates to see if they can explain all conflicts. This method could be applied in our diagnosis strategy by extending the present hypotheses, from a candidate space, to explain multiple faults when new observations are reported by the user that are abnormal and not explained by the present hypotheses. For example, we could extend the second clause of program (5-17) by modifying the conjunct *sort_hypotheses* to *sort_and_extend_hypotheses*, that would implement this method.

5.6 SUMMARY

To summarize this chapter, we have discussed how a representation of a diagnosis strategy can be separated from the knowledge of a particular domain. The knowledge was represented partly as an object theory and partly as domain specific compiled knowledge in the relations *meta_refute* and *probably_refutable* in a metatheory. The diagnosis strategy was represented in a metatheory as a logic program.

Chapter 6

An Example Diagnosis Case

Let us now investigate an example computation of a diagnosis, using program (5-16) of the previous chapter, to see how the diagnosis system can work in practice. The point of view will be the internal view of the diagnosis system, so we will see the results of the computation at various points of the diagnosis. All interaction with the user will be shown in a simplified form, but the actual interaction between the user and the system will be discussed in the next chapter. A prototype of the system has been implemented by the author.

6.1 INTRODUCTION

We picture a situation where we have a user of a stereo system who is not fully confident in its use. In particular, the user is not capable of diagnosing malfunctions. Moreover, it has been delivered with a computer based diagnosis system and a simple device for measuring electrical voltage. The basis for this example will be the object theory discussed in Chapter 4 together with the general parts of the metatheory, two clauses of *meta_refute*—programs (5-9), (5-12) and one clause of *probably_refutable*, program (5-13).

In section 5.4.3 we discussed how rules such as the *probably_refutable* clause could be the basis for sorting the hypotheses before selection of which hypothesis to test. In this example we assume a basic sorting order for hypotheses ranking settings and connection hypotheses before intern hypotheses, because settings and connections are easier for the user to observe than internal signals. In conclusion, the basic sorting order will first be used to sort the hypotheses, then the *probably_refutable* clause will be used for further sorting.

6.2 THE COMPUTATION OF A DIAGNOSIS

We now know what metalevel rules will be used in this example, let us discuss the actual computation. As we saw in program (5-16), the first step of the diagnosis strategy is that the user

indicates which domain to diagnose, by selecting an object theory. In this example, the user of course selects *stereo*. As the next step, the user indicates what the problem is with the stereo, which we assume in this case is that the sound from the loudspeakers is not as expected. So, the user indicates that the sound of the loudspeakers is bad. The user also indicates in which mode this problem appears and in what type of configuration the components are connected. After this, the predicate *select_observation* succeeds and the system now has its first observations.

Obs = [behaviour(s1, [loudspeaker,sp1], cd_player, sound, bad),
behaviour(s1, [loudspeaker,sp2], cd_player, sound, bad)]

These observations mean that the loudspeakers do not sound ok, at least not when the stereo is used in CD-player mode and the components are connected in configuration *s1*. In the next step, the diagnosis system selects an observation to generate hypotheses from. This choice can of course be based on domain specific factors, but in this example the system simply selects the first observation *behaviour(s1, [loudspeaker,sp1], cd_player, sound, bad)*. Before any hypotheses are generated the strategy checks that the observation does not correspond to a statement of the object theory—the system checks that the observation is abnormal with respect to the object theory *stereo*. After this check the *hypotheses* program (5-4) generates hypotheses for the selected observation. In this example, the program generates a list *Hs* of 19 hypotheses that may have caused the loudspeaker problem. The hypotheses concern three groups of problems—internal faults in an object, faulty settings of a component and incorrect connections between objects.

1. *intern([loudspeaker, sp1], s1, cd_player),*
2. *intern([music_cable(loudspeaker), w6], s1, cd_player),*
3. *intern([amplifier, am], s1, cd_player),*
4. *intern([power_cable(amplifier), w8], s1, cd_player),*
5. *intern([power, acdc(w8)], s1, cd_player),*
6. *intern([music_cable(cd_player), w1], s1, cd_player),*
7. *intern([cd_player, cd1], s1, cd_player),*
8. *intern([power_cable(cd_player), w12], s1, cd_player),*
9. *intern([power, acdc(w12)], s1, cd_player),*
10. *settings([amplifier, am], s1, cd_player),*
11. *settings([cd_player, cd1], s1, cd_player),*
12. *conns(s1, [loudspeaker, sp1], [music_cable(loudspeaker), w6]),*
13. *conns(s1, [music_cable(loudspeaker), w6], [amplifier, am]),*
14. *conns(s1, [amplifier, am], [power_cable(amplifier), w8]),*
15. *conns(s1, [power_cable(amplifier), w8], [power, acdc(w8)]),*
16. *conns(s1, [amplifier, am], [music_cable(cd_player), w1]),*
17. *conns(s1, [music_cable(cd_player), w1], [cd_player, cd1]),*
18. *conns(s1, [cd_player, cd1], [power_cable(cd_player), w12]),*
19. *conns(s1, [power_cable(cd_player), w12], [power, acdc(w12)])*

After the generation of hypotheses, the predicate *diagnosis* is responsible for the rest of the diagnosis—for finding a diagnosis among these hypotheses. The first step of the predicate is to evaluate the potential questions connected to the *meta_refute* clauses to see if a question should be put to the user. In this example, the question (5-13) connected to the first *meta_refute* clause (5-9) will be put to the user, since it may be able to refute five of the 19 hypotheses.

"Please indicate some normal activity in the stereo."

Suppose the user indicates that the on-light of the amplifier is on. The following observation will be appended to the previous user observations *Obs* giving *Obs2*.

behaviour(s1, [amplifier, am], cd_player, [light, on(am)], on)

After this user interaction, the *diagnosis* predicate tries to do a meta refutation from *Obs2* of each of the five hypotheses found above to be potentially refutable. The result is that the *meta_refute* clause (5-9) refutes the five hypotheses below.

4. intern([power_cable(amplifier), w8], s1, cd_player)
5. intern([power, acdc(w8)], s1, cd_player)
9. intern([power, acdc(w12)], s1, cd_player)
14. conns(s1, [amplifier, am], [power_cable(amplifier), w8])
15. conns(s1, [power_cable(amplifier), w8], [power, acdc(w8)])

The system now has 14 hypotheses left to investigate. The *diagnosis* predicate proceeds to sort the hypotheses in the basic sorting order, thus *settings* and *conns* hypotheses will be first, then the *intern* hypotheses will follow: 10, 11, 12, 13, 16, 17, 18, 19, 1, 2, 3, 6, 7, 8

The *probably_refutable* clause (5-13) is not true given the present set of observations, so it will not affect the sorting. Next in *diagnosis*, a hypothesis will be selected by *selected_hypothesis*. In this example, the predicate simply selects the first hypothesis, i.e.,

10. settings([amplifier, am], s1, cd-player).

The *diagnosis* predicate now attempts a meta refutation of this hypothesis to see if a rule can refute the hypothesis, but this does not succeed since no *meta_refute* rule is true for this hypothesis. So, an object refutation is attempted. This method of refutation uses the object theory as a basis for the refutation. The first step of the object refutation program (5-6) is to find a refutation for the hypothesis, and *settings([amplifier, am], s1, cd_player)* matches the second clause of the *refutation* program (5-5), which derives the following statements from the object theory.

setting(s1, [amplifier, am], cd-player, [button, speaker(am)], a)
setting(s1, [amplifier, am], cd-player, [button, on(am)], on)

The second step of the method is to verify that these statements can be observed by the user—if they are verified the refutation succeeds otherwise the method fails. At this point it is not known to the system whether these statements can be observed or not. Therefore, the system puts a question to the user regarding each. Let us assume the user verifies these statements as observable. The hypothesis will then be refuted. It cannot be the case that the cause of the problem is the settings of the amplifier since the user can observe the correct settings in his stereo.

The last step of the *diagnosis* predicate is to continue to search for a diagnosis by recursive calls to itself, because the selected hypothesis was refuted. So, the predicate will be repeated with the 13 hypotheses that remain until one is found for which the user indicates that some statement for the hypothesis can not be observed.

The next hypotheses to investigate will be

12. conns(s1, [loudspeaker, sp1], [music_cable(loudspeaker), w6])
13. conns(s1, [music_cable(loudspeaker), w6], [amplifier, am])

because the *probably_refutable* clause is now true. The system has investigated the hypothesis about the settings of the amplifier, so the conditions in the rule are now true. Instead of hypothesis 11 that was the next hypothesis, numbers 12 and 13 are selected.

So, the system investigates hypotheses 12 and 13 and first asks the user about the connections between the loudspeaker and the cable to the loudspeaker and then about the connections between that cable and the amplifier. Let us assume the user verifies these connections so that these two hypotheses are also refuted.

The *diagnosis* predicate will then continue to investigate the 11 hypotheses not yet tested. The hypotheses will be investigated with *object_refute*, since no *meta_refute* clauses will yet be applicable. The remaining hypotheses are 11, 16, 17, 18, 19, 1, 2, 3, 6, 7, 8 and the first five investigated are:

11. settings([cd_player, cd1], s1, cd_player)
16. conns(s1, [amplifier, am], [music_cable(cd_player), w1])
17. conns(s1, [music_cable(cd_player), w1], [cd_player, cd1])
18. conns(s1, [cd_player, cd1], [power_cable(cd_player), w12])
19. conns(s1, [power_cable(cd_player), w12], [power, acdc(w12)])

Assume the user verifies each of the statements connected to each of the above five hypotheses. At this point we have only *intern* hypotheses left to investigate (1, 2, 3, 6, 7, 8). Since the question (5-15) for the *meta_refute* clause (5-12) has precisely this as a condition for its use, it has not been asked yet. However, that question will now be asked.

"Please indicate some abnormal signal in the stereo."

Let us assume the user answer indicates that the output signal of the music port on the CD player is measured to 200 mV. This is not a normal value for this port, so the *meta_refute* clause will be true. Therefore, the five hypotheses below are now refuted.

1. intern([loudspeaker, sp1], s1, cd_player)
2. intern([music_cable(loudspeaker), w6], s1, cd_player)

3. intern([amplifier, am], s1, cd_player)
6. intern([music_cable(cd_player), w1], s1, cd_player)
8. intern([power_cable(cd_player), w12], s1, cd_player)

After the meta refutation of these five hypotheses there is only one remaining.

7. intern([cd-player, cd1], s1, cd-player)

This hypothesis cannot be refuted since one statement for it is an output signal of the music port of 400 mV, which is clearly not equal to the user's observation of 200 mV given previously.

So, the result is that the fault lies in the internal function of the CD player *cd1*, i.e. the diagnosis is *intern([cd_player, cd1], s1, cd_player)*.

Chapter 7

Interacting With the User

The purpose of our diagnosis system is to identify malfunctioning components or an incorrect setup of the components in the domain. By asking the user about the behaviour of the components, the diagnosis system can collect symptoms from which conclusions about other components can be drawn. These questions must be put to the user through a *user interface*. In this chapter we will discuss the design and representation of a user interface for a domain suited for visual illustration.

7.1 PROPERTIES OF A USER INTERFACE

What demands can be put on a user interface? At least the following demands can be stated: (1) the design of the user interface should be clear and easy to understand for the user, (2) it should be possible for the system to express questions and receive answers from the user through the interface, (3) it should be possible to express explanations to the questions and conclusions of the diagnosis system through it, (4) it should be easy to modify and (5) it should be represented in such a fashion that it can be integrated with the diagnosis system.

We can distinguish two ways of interacting with the user of a system: the first is text-based interaction (or linear interaction) and the second is visually based interaction (or n-dimensional, where n>1). Examples of text-based interaction can be found in the early expert systems in which questions to the user were put and answered on a text terminal. The following text-based dialogue from the MYCIN system in [17] illustrates this kind of interaction. In this dialogue, the user's answers are preceded by asterisks (**).

...

(4) Have you been able to obtain positive cultures from a site at which Fred Smith has an infection?

**YES

----------INFECTION-1------------

(5) What is the infection?

**PRIMARY-BACTEREMIA

(6) Please give the date and approximate time when signs or symptoms of the primary-bacteremia (INFECTION-1) first appeared (mo/da/yr)

**MAY 5, 1975

The most positive culture associated with the primary-bacteremia (INFECTION-1) will be referred to as:

---------CULTURE-1----------

....

Let us discuss some disadvantages with text-based interaction compared to visually based interaction. First, in complex domains it may be difficult for the user to know *which object in the domain* a term in a text refers to. A case in point would be a text-based dialogue for diagnosis of the stereo system discussed in this thesis, where it would be inconvenient to identify in text various parts of the components. For example, if the system were to ask about a connection to the amplifier, a text could be "Is the second port of the loudspeaker ports on the amplifier connected to the minus pin of the loudspeaker cable?" It is not entirely obvious for a naive user of the stereo system which of the ports of the amplifier is the loudspeaker port or even which of the components is the amplifier, when the system asks this question. Visually based interaction offers more possibilities than text-based interaction. With a visual illustration in which the system points to objects when asking questions, the user can rely on recognition of the objects in the domain rather than being forced to remember their names. A visual illustration of the domain through which the user and the system could communicate, e.g., a simple picture of the components in the stereo, would be sufficient for this. Furthermore, the visual mode of interaction affords *direct manipulation* of the objects in the illustration. The user could thus take the initiative in the dialogue and manipulate the illustration directly to provide information to the system. For instance, the settings of a component could be given in this way. Second, in text-based interaction the user must remember the *correct keywords for each action* if the computer is to respond as desired. In contrast, visually based interaction can use menus of commands and dialogue boxes with push buttons to guide the user in how to start actions. Third, it can be *difficult to formulate texts* that are sufficiently clear to convey the intended meaning. This is a matter of formulating unambiguous texts, which is not always so easy. This problem remains to some degree with visual interaction because texts must be formulated also in this approach. However, since much of the interaction is visual, the need for texts is smaller and the texts that are needed can be given visual support with graphical symbols connected to the text. The system can for instance display a warning sign when a particularly important message is given or a picture of the measuring device can be displayed when a measurement is required to give the user a visual cue. Finally, *explanations* can be given visual support in the illustration. By highlighting various parts of the graphics, the system can illustrate how the domain works to give the user an understanding of why a question has been asked or how a conclusion has been reached. The visual illustration of the structure of the domain gives the user more support than it is possible to provide in a text-based explanation.

In the case of the stereo domain it is quite obvious that a graphical illustration would be beneficial for the communication between the system and the user because the objects in the domain are indeed very visual. They are easily recognizable for the user. Consider, on the other hand, the domain of MYCIN—diagnosis of disorders in the blood. In this domain it is not obvious that a graphical illustration is superior to a text-based dialogue since the objects in the domain are not as visually distinguishable as in the stereo domain. An infection is not very visual and therefore not very easy to illustrate. The choice of mode of interaction is highly dependent on the character of the domain.

Visually based ways of interaction can be ordered on a scale depending on the complexity of the illustration, from simple two-dimensional graphical illustrations to three-dimensional illustrations with colour, sound and movement in a virtual reality. Leftmost on the scale, a simple two-dimensional picture can be created in an ordinary drawing program. The next step on the scale could be moving pictures that would illustrate some feature of the domain. To further enhance the presentation, it would be possible for the user to control the presentation, for example from what direction and distance the image is viewed. Finally, in a virtual reality presentation the user would move about freely in a representation of the domain and manipulate its objects. For example, the user would be able to turn objects around, operate objects—perhaps turning a cd-player on—or open objects to investigate their interior. We see that the possibilities for visual illustration are indeed many. In this thesis, however, we will only discuss how a simple two-dimensional graphical display can be used for interaction between the user and the stereo diagnosis system, in order to give an idea of the potential of visually based interaction. The design of more advanced visual illustrations requires further study and is beyond the scope of the present investigation.

7.2 DESIGN OF A GRAPHICAL USER INTERFACE FOR A DIAGNOSIS SYSTEM

A basic motivation for using a two-dimensional graphical display as a basis for a user interface is to provide the user with a visualization of the domain, in which the user can identify different parts. For the stereo system represented by the axioms of Chapter 4, we would need to illustrate the different stereo components and their subparts, the cables between the components and how the cables are connected to the components. In Fig. 7-2 below we find an example of how a (part of a) graphical interface for a diagnosis system can be designed. This picture corresponds to the stereo system of Chapter 4. We see the components (from the front and from the back), the cables and the connections between them.

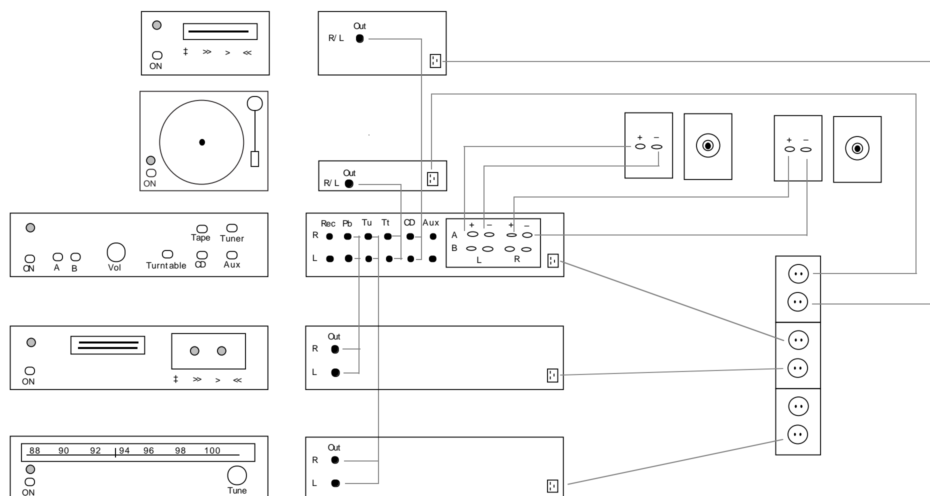


Fig. 7-1: *A graphical display as a user interface for a diagnosis system.*

In this picture the diagnosis system can express questions to the user by displaying a box, containing a question, pointing to the part of the stereo system the question refers to. The user can answer the question positively by clicking on a “yes-button” and negatively by clicking on a “no-

button". By clicking on the "other-button" the user could start a dialogue with the system to get further information, perhaps ask for an explanation of why the question is asked. In Fig. 7-2 we see an example of how a question can be put to the user. By letting the dialogue box point to the part of the stereo

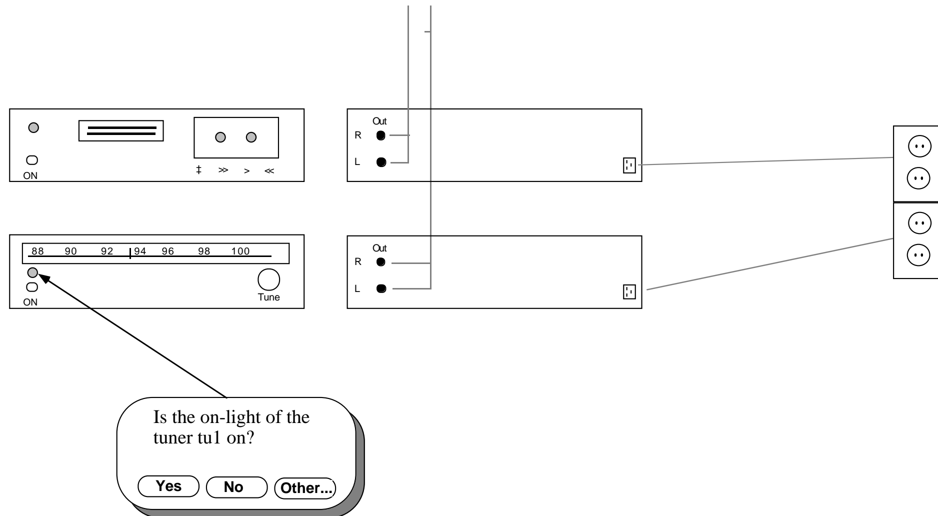


Fig. 7-2: *Asking a question to the user through the display.*

system the question refers to, the diagnosis system makes it easier for the user to identify it.

Alternatively, the user could be given the initiative in the dialogue and volunteer information about the diagnosis case. With this style of interaction the user would click on parts of the picture to indicate which part of the domain he would like to enter information about.

If we design the picture so that parts of the it could change, then the user would be able to indicate the settings of the stereo system by adjusting items in it. For example, if the user clicked on the on-button of a component, the picture would adjust to reflect that it is pressed and the system would know its status. This is an example of how graphical displays can provide direct manipulation interaction.

To speculate further, it could be possible to use the graphical picture when expressing explanations of the diagnosis system's reasoning. One possibility would be to illustrate, e.g. how the power to the stereo system is supplied to various components. A why-question would be: "Why is the question about the on-light of the tuner asked?" To answer this question the system would emphasize relevant parts of the graphical picture. In Fig. 7-3 below, we find an example why-explanation. The cable from the power supply to the tuner is marked in heavy black-and-white and the on-light of the tuner flashes. It indicates that a positive answer to the question regarding the on-light would make it possible to deduce that the power supply to the stereo system is OK. The graphical display would amplify a text-based explanation. However, this kind of visual explanation requires that we represent information about how the objects of the stereo system appear visually in different situations and metalevel knowledge about the stereo domain to give reasonable explanations. The axioms in the object theory would be the basis for the explanation. The problem of generating explanations for the system's behaviour will not be discussed further in this thesis.

These examples illustrate how a picture can be used as a basis for an interface in a diagnosis system. In the next section we will discuss how we can represent some of the knowledge needed for this

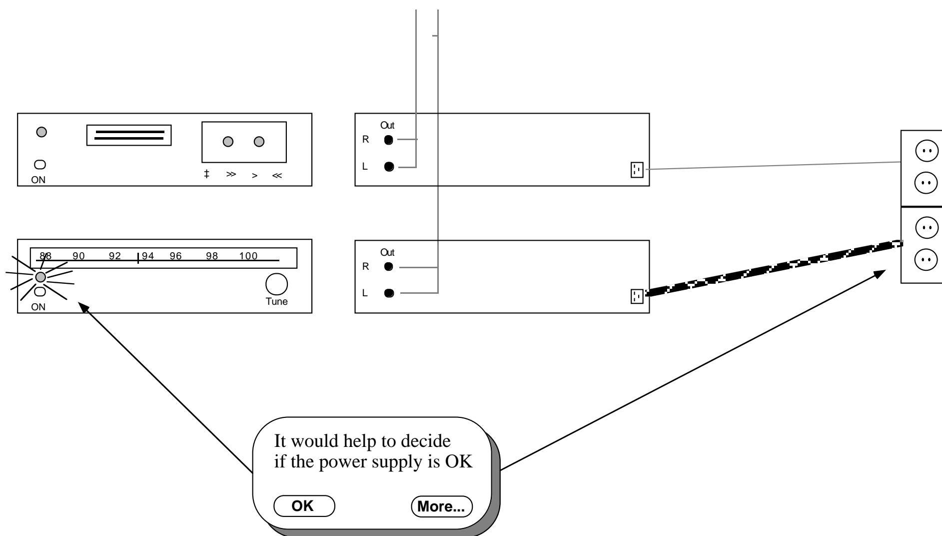


Fig. 7-3: *Illustrating an explanation in the graphical picture.*

kind of graphical interface, in particular regarding the connection between the picture and the object theory.

7.3 A REPRESENTATION OF KNOWLEDGE FOR A GRAPHICAL USER INTERFACE

In the human-computer interface literature it is often argued that the user interface of a software system should be represented separately from the rest This quality of a software system is known as *dialogue independence*. To achieve dialogue independence it is recommended to divide an interactive application system into two components: a dialogue component and a computational component. The *dialogue component* manages the communication with the end-user of the system, whereas the *computational componen*t implements the functional processing mechanisms. The communication between the two components is carried out through an *internal interface*, which integrates them. The motivation for dialogue independence is that the design of dialogue should be separate from the design of computational software, so that changes in one part can be made without changing the other. This makes it possible to design and re-design each component independently of the other (provided that each is consistent with the internal interface). As a result the process of designing the interface and the computational software will be flexible. [28]

In this section we will discuss how dialogue independence can be achieved in the diagnosis system. In particular we discuss the internal interface of the system. This issue is related to the discussion about user interface management systems on page 54. Other aspects regarding the design of a user interface is beyond the scope of this thesis.

The computational component of the diagnosis system would consist of the metatheory and the object theory and the dialogue component by the part of the diagnosis algorithm that puts questions to the user about observations in the diagnosis domain. The internal interface between these components would communicate the statements of the object theory. However, since we have a graphical display as a basis for the user interface we must provide a connection between the statements of the object theory and the display. One approach to such a connection is to represent the relationship between the objects of the statements and the picture. We would then be able to draw the picture in an ordinary drawing

program and represent the relationship between the objects of the object theory and the corresponding parts in the picture as a relation. What kind of parts in a graphical display would we need to represent? We have, basically, two kinds of pictures that we would need to represent for a stereo system: rectangular pictures and straight line pictures. The first kind of picture can be represented with two pairs of numbers for the picture: an (X,Y) coordinate that represents the upper left corner position of the object in the picture and a pair (Height, Width) that represents the size of the object in the picture.

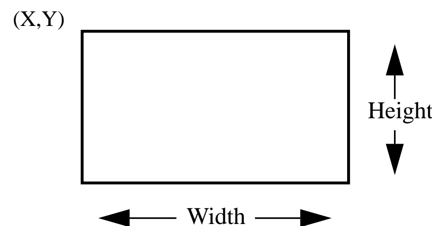


Fig. 7-4: *A rectangular picture can be represented as two pairs of numbers.*

The second kind of pictures—line pictures—needs a sequence $[(X_1,Y_1),(X_2,Y_2), ..., (X_n,Y_n)]$ of points of reference since the line can change direction in the middle. (X_1,Y_1) and (X_2,Y_2) represents the first segment of the line, (X_2,Y_2) and (X_3,Y_3) the second segment, and so on. The line also needs a width reference Width.

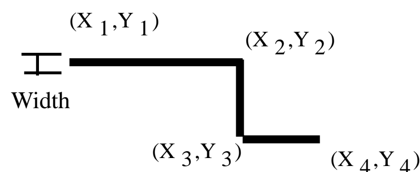


Fig. 7-5: *A line picture can be represented as a sequence of pairs and a width.*

We can then represent the relationship between such parts of the picture and the objects of the object theory as a relation *position*, between a name for a stereo configuration, a name for an object, a position reference in the picture and a size reference.

Example:
position(s1, [amplifier, am], (317,217), (68,276))
position(s1, [power_cable(amplifier), w8], [(587,269), (835,353)],1)

The above clauses represent the positions of the amplifier *am* and the power cable *w8* in Fig. 7-2. Note that the y-coordinates increase downwards, and the x-coordinates increase rightwards. This relation is the basis for the internal interface between the dialogue component and the computational component of the diagnosis system. By using names for the objects of the object theory in the relation *position*, we can locate the illustration of each object in the picture. So, the name for an object is the connection between the metatheory and the interface. The connection works when we want to illustrate a question regarding an observation with a question-box in the picture, since the object theory objects are used in

the statements of the object theory. For example, the system may need to ask the user whether the following observation is true:

signal(s1, [cd_player, cd1], cd-player, [mu_port, out(cd/rl, cd1)], 400 mV)

We then know the location of the illustration of the music port in the picture by using the name for the object *[mu_port, out(cd/rl, cd1)]* and the name for the configuration *s1* to find the coordinates of the object in the picture via the relation *position*. By calculating the boundaries of the object as defined in *position* we can determine where in the picture the object is illustrated and where the question box should be displayed. The same method works when the user clicks on a point of the picture to indicate an object, since the coordinates of the click can be matched with the coordinates in *position* to find out which object the user clicked on.

In using a drawing program to construct the graphical display for the interface and the *position* relation for the connection between the picture and the objects of the object theory, we have a a straightforward method. Modify the picture in the drawing program and save the picture as file, then load the file into the programming system and modify the coordinates of the components in the relation *position*. It is easy to modify the interface and it is well integrated with the diagnosis system since the names for the objects of the object theory are used in the interface.

With this kind of representation it would be possible to have multiple pictures for different types of illustration, e.g., a graphical picture and a photograph. The user could choose with which picture to interact. To represent different pictures for a domain we extend the relation *position* with the name for each picture.

Example:
position(drawing, s1, [amplifier, am], (317,217), (68,276))
position(photo, s1, [amplifier, am], (412,240), (53,264))

An alternative approach to the representation of pictures can take advantage of a graphical subsystem, such as MacProlog's graphic windows [33]. This system can manage an abstract database of graphic objects which we 'add' to a graphic window associated with the abstract database. In MacProlog it is possible to describe the structure and appearance of graphics objects as terms in a specific graphic description language (GDL). Since the system manages an abstract database associated with a graphic window it can determine which object the user has clicked on, as well as 'add' further pictures in relation to existing pictures in the graphic window. We can use the graphic subsystem of MacProlog to implement a graphical interface, such as the example in Fig. 7-2. However, if we would like to use external pictures, such as a photograph, we would still need the first approach discussed above.

To summarize, we have discussed some aspects of a representation of a graphical user interface that can be modified if we need to change a picture. With the approach discussed above only the relation *position* needs to be modified correspondingly when a picture is altered. This approach allows us to have multiple pictures of the domain, which can be suitable for different aspects of the diagnosis process.

Chapter 8

Future Work

In this chapter we discuss future developments of the ideas in the thesis. Two areas related to each other will be considered—the potential for modifying the diagnosis system to other domains and the possibilities for supplying computer support for the process of constructing such diagnosis systems.

8.1 GENERALIZATION OF THE DIAGNOSIS SYSTEM

The long-term goal of this work is to construct shells for developing systems integrating heuristic and model-based diagnosis, for different classes of domains. Instead of starting from scratch when developing a diagnosis system, spending time to design appropriate knowledge representation structures and inference mechanisms, a knowledge engineer would use a shell. The knowledge engineer would fill the shell's representation structures with domain knowledge and adjust the inference mechanism to suit the particular needs of the new system. The shell would ideally provide computer support for this process, as will be briefly discussed in section 8.2.

The idea of general software for expert systems is not new. The first expert system shell was EMYCIN—'Empty MYCIN'— which used the representation form and the inference mechanism of MYCIN. Numerous other expert system shells have been developed and become quite popular. In the model-based approach, GDE can be considered an expert system shell with an inference mechanism (ATMS) used with models for various components of circuits.

In order to develop a shell for a class of domains we must identify the types of objects common between domains and apply this to some other domain(s). The objects used in the formalization of the stereo domain presented in Chapter 4 could be used in the formalization of other domains—components, connectors and connections could fit as abstractions of electronic circuits or other systems. Settings, signals and behaviours of these objects could also occur in such domains. As an example, we consider a monitoring and control system for a machine on an assembly line system.

MACS II (Monitoring And Control System) is a computer based system for monitoring and controlling the process of tightening threaded joints. The MACS system is used, for example, on

assembly lines of motor vehicle manufacturers where the tightening of joints for motor blocks must be rigorously monitored and controlled. It comes in different configurations and consists of up to 30 spindle nutrunners that tighten joints, controlled by electronic boards (spindle modules) connected to a system module and opto relays converting digital signals to analogue. Additional components (some optional) include a voltage aggregate for power supply, pre-amplifiers for amplifying signals between the spindle modules and the multiples controlling the spindles, and finally, printer/terminal for the operating staff to communicate with the system. In Fig. 8-1 below we see an outline of an example MACS systems.

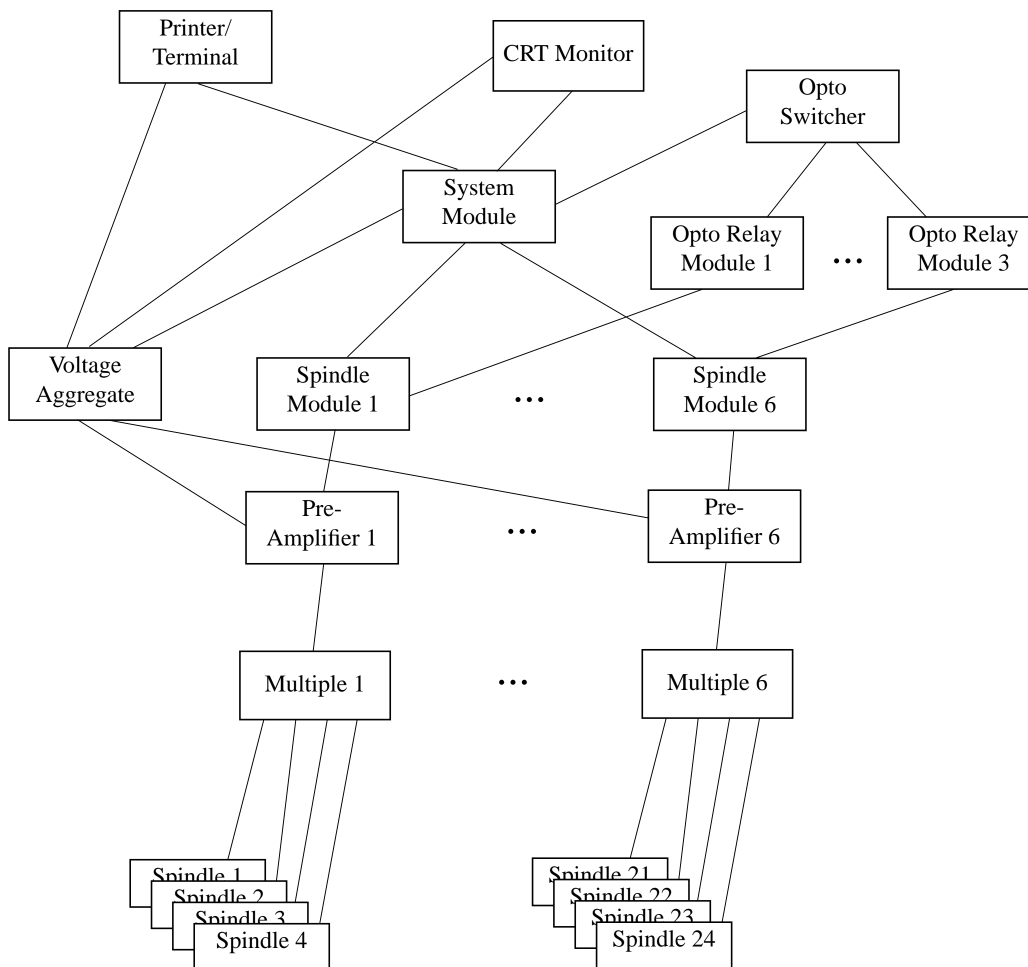


Fig. 8-1: *An outline description of an example MACS II system.*

The structure and function of these components and connections could be formalized as an object theory similar in structure to the theory presented in Chapter 4. In outline, we can view the MACS system as a set of components (spindle modules, spindles, pre-amplifiers, etc.) connected to each other, communicating with measurable signals through the connectors (angle and torque signals, control signals etc.) each having observable behaviour (LED's, printouts, screen messages etc.) With such a formalization of the domain, the diagnosis theory of Chapter 5 could be adapted to perform diagnosis of the MACS system since the diagnosis theory utilizes the structure of the object theory for its function. Naturally, the heuristics and hypothesis-generating parts of the diagnosis theory would

have to be modified for the new domain, but the major definitions could be re-used. The stereo domain and the MACS domain would be two instances of the class of domains that are characterized by objects connected with some type of directed flow.

The correct function of a MACS system is vital for a production site since if it halts, the entire production line will come to a halt, which will cost a lot of money. Therefore, a diagnosis procedure is important for the users of MACS. There exists a diagnosis handbook for the MACS system, to make it possible for the operating staff to diagnose the system to some extent. The diagnosis handbook for the MACS II system consists of fifteen fault trees that cover fifteen major faults, together with other illustratory material. This handbook is intended for the staff operating the production line in the factory and if a more complex error appears a technician from the manufacturer is requested. A computerized diagnosis system would probably extend the range of faults that could be diagnosed by the operating staff. Below we see an example fault tree for the MACS II system covering the fault ‘The system does not start’. It consists of a line of questions that should be answered by the operator to identify the cause of the fault and repair it.

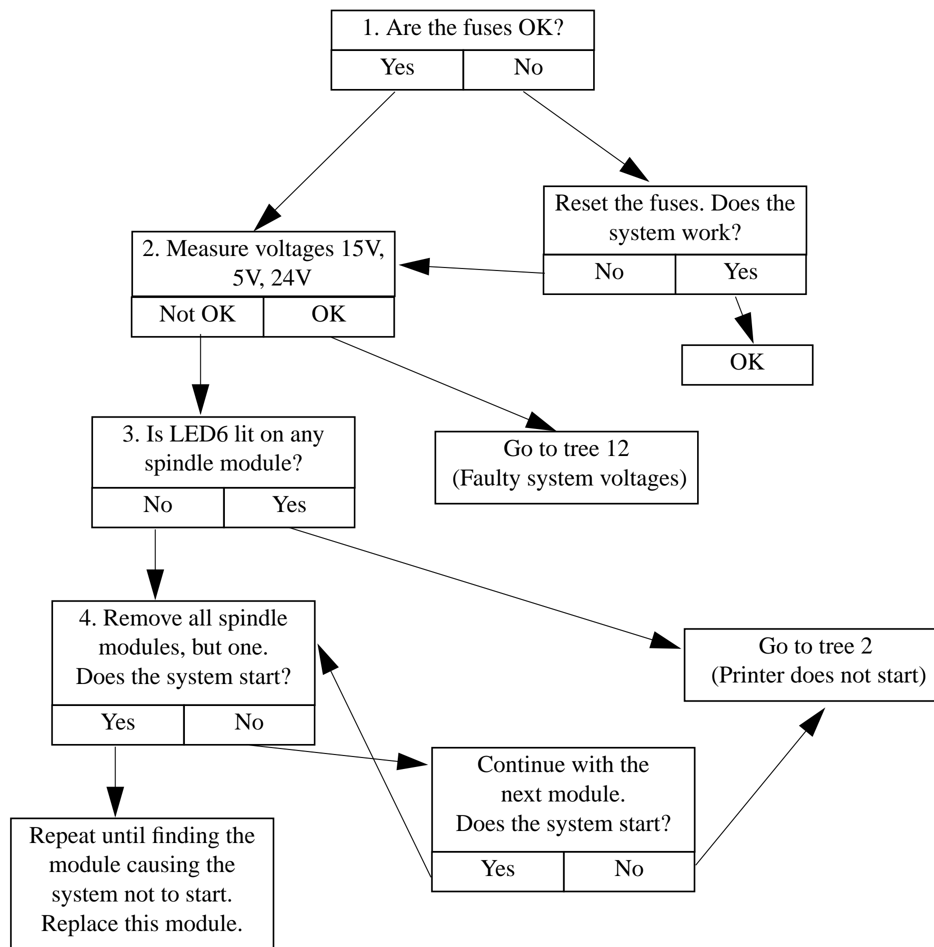


Fig. 8-2: *A fault tree for the MACS II system.*

The fault tree represent an example of compiled knowledge, there is no information—except implicitly—what the principles of function in the MACS system underlying this fault tree are. It would probably be difficult to re-use this knowledge for constructing a diagnosis handbook for another, similar

machine, since the underlying principles are hidden. In a model-based computerized diagnosis system of the kind proposed in this thesis this would be represented as explicit knowledge in an object theory. The parts of the diagnosis knowledge that either would be below the level of abstraction chosen for the theory or had a heuristic character, would be represented in the metatheory. However, an implementation of such a diagnosis system for MACS systems remains to be investigated.

8.2 COMPUTER SUPPORT FOR THE CONSTRUCTION OF DIAGNOSIS SYSTEMS

The lack of computer supported modelling techniques has been recognized as one of the major bottlenecks in applying model-based reasoning to real problems [16] and there does not seem to be any evidence that model-based diagnosis is different. One approach to provide computer support for modelling of domains could be to use a schematic representation of models for a class of domains and formalize a theory to aid the instantiation of such a representation.

In the author's view, the integration of heuristic and model-based reasoning could alleviate some of the burden to construct accurate models in some domains, since some parts of a domain that are difficult to model at an effective level of abstraction, could be described in a metatheory as compiled knowledge. If it is difficult to correctly model some phenomena in detail it may be easier to resort to compiled knowledge in a metatheory for those parts of the domain, but still model other parts in the object theory.

Let us, however, briefly discuss how a computer support for construction of diagnosis systems could be designed. The knowledge engineer would choose a diagnosis shell for the class of domains appropriate for the diagnosis system in mind. A shell would consist of a schematic representation of a diagnosis system such as the one described in Chapter 4 and Chapter 5. This would include schemata or templates, for the formulas of the object and metatheory. The instantiation of these schemata would be controlled by a meta-metatheory that formalized the instantiation process. To aid the instantiation of the schemata the knowledge engineer could use a graphical interface in which components, connectors and their connections would be specified instantiating the structure-describing parts of the object theory. Some components could be standard components with a pre-specified function of the component. When specifying the structure of the domain the knowledge engineer would automatically specify the function of the objects in the domain. The function of non-standard components would, perhaps, have to be formalized manually by the knowledge engineer. Another open question is how to provide support for the instantiation of the schemata of the metatheory.

Chapter 9

Conclusions

The thesis has studied the problem of integrating model-based and heuristic diagnosis in a single system. The goal has been to give a representation of knowledge such that its logical structure is preserved, it is represented in a modular fashion allowing parts to be replaced or modified without affecting other, the representation is transparent enough for the knowledge engineer to distinguish different kinds and the representation allows them to be used together.

By introducing a metalogic representation structure with a metatheory and an object theory the above goals have been achieved for the diagnosis example that was investigated. The logical structure of knowledge is preserved by the division of the metatheory and object theory where strategic and heuristic knowledge is represented in the metatheory and principled (model-based) domain knowledge is represented in the object theory. A modular representation of the knowledge in the diagnosis system is likewise achieved by this division, where the theories are the modules. The foundation for a transparent representation is provided by the meta and object theory representation structure which allows the knowledge engineer to represent different kinds of knowledge separately. Finally, the integration of model-based and strategic/heuristic knowledge via the refutation methods in the metatheory allows them to be used together.

We have argued that the integration of model-based and heuristic diagnosis is interesting for the following reasons—usefulness of heuristics and problems of representing complete models in many domains. In constructing the representation structure it has been demonstrated that it is possible to integrate model-based and heuristic diagnosis. The integration has indicated how powerful inferences can be represented in the metatheory, i.e. semi-general metarules stated in terms of an object theory.

It is also argued that the design of a graphical user interface should support the user in giving observations. Finally, we conclude that it would be desirable to have diagnosis shells for classes of domains that support the construction of integrated heuristic and model-based diagnosis systems.

References

1 K. Andersson, Reflection in Prolog, working paper (Uppsala: Uppsala University, Upmail, 1990).
2. K.A. Bowen and R.A. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, in: K.L. Clark and S.-Å. Tärnlund (eds.), *Logic Programming* (London: Academic Press, 1982).
3. J.S. Brown, R.R. Burton and J. de Kleer, Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II, and III, in: D. Sleeman and J.S. Brown (eds.), *Intelligent Tutoring Systems* (New York: Academic Press, 1982) 227–282.
4. R. Carnap, *The Logical Syntax of Language* (New York: Harcourt, Brace; London: Kegan Paul, Trench, Trubner, 1937).
5. M.A. Carrico, J.E. Girard and J.P. Jones, *Building Knowledge Systems* (New York: McGraw-Hill, 1989).
6. B. Chandrasekaran, Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks, *Proc. Tenth International Joint Conf. on Artificial Intelligence*, Milano (1987) 1183–1192.
7. W.J. Clancey, Heuristic Classification, *Artificial Intelligence* 27 (1985) 289–350.
8. W.J. Clancey and R. Letsinger, NEOMYCIN: Reconfiguring a Rule-Based Expert System for Application to Teaching, *Proc. Seventh International Joint Conf. on Artificial Intelligence*, Vancouver, Canada (1981) 829–836.
9. K.L. Clark, Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, (New York: Plenum Press, 1978) 293–322.
10. A. Colmerauer, H. Kanoui, M. van Caneghem, R. Pasero and P. Roussel, *Un Système de Communication Homme-Machine en Français* (Marseille: Univ. d'Aix-Marseille II, 1973).
11. L. Console, D. Dupré and P. Torasso, A Theory of Diagnosis for Incomplete Causal Models, *Proc.*

Eleventh International Joint Conf. on Artificial Intelligence, Detroit (1989) 1311–1317.

12. L. Console and P. Torasso, A Logical Approach to Deal with Incomplete Causal Models in Diagnostic Problem Solving, *Lecture Notes in Computer Science* 313 (Berlin: Springer Verlag, 1988) 255–264.
13. L. Console and P. Torasso, Integrating Models of the Correct Behaviour into Abductive Diagnosis, *Proc. 9th European Conf. on Artificial Intelligence*, Stockholm (1990) 160–166.
14. S. Costantini and G.A. Lanzarone, A metalogic programming language, in: G. Levi and M. Martelli (eds.), *Proc. Sixth International Conf. on Logic Programming*, Lisbon (Cambridge, Mass.: MIT Press, 1989) 218–233.
15. P.T. Cox and T. Pietrzykowski, General Diagnosis by Abductive Inference, *IEEE Symposium on Logic Programming* (1987).
16. P. Dague, Qualitative Reasoning: A Survey of Techniques and Applications, *AI Communications* 8 (1995) 119–192.
17. R. Davis, B. Buchanan and E. Shortliffe, Production Rules as a Representation for a Knowledge-Based Consultation Program, *Artificial Intelligence* 8 (1977) 15–45.
18. J. de Kleer, An assumption-based truth maintenance system, *Artificial Intelligence* 28 (1986) 127–162.
19. J. de Kleer and B.C. Williams, Diagnosing Multiple Faults, *Artificial Intelligence* 32 (1987) 97–130.
20. J. de Kleer and B.C. Williams, Diagnosis with Behavioral Modes, *Proc. Eleventh International Joint Conf. on Artificial Intelligence*, Detroit (1989) 1324–1330.
21. B. Faltings, Working group in model-based design and reasoning. Part I: modeling and diagnosis, *AI Communications* 9 (1996) 59–64.
22. M.R. Genesereth, The use of design descriptions in automated diagnosis, *Artificial Intelligence* 24 (1984) 411–436.
23. M.R. Genesereth and N.J. Nilsson, *Logical Foundations of Artificial Intelligence* (Palo Alto: Morgan Kaufmann Publishers, 1987).
24. T.R Gruber, *The Aquisition of Strategic Knowledge* (San Diego: Academic Press, 1989).
25. W. Hamscher, Modeling digital circuits for troubleshooting, *Artificial Intelligence* 51 (1991) 223–271.
26. W. Hamscher, L. Console and J. de Kleer (eds.), *Readings in Model-Based Diagnosis* (Morgan Kaufmann, 1992).
27. A. Hart, *Expert systems: an introduction for managers* (London: Kogan-Page, 1988).
28. H.R. Hartson and D. Hix, Human-Computer Interface Development: Concepts and Systems for its Management, *ACM Computing Surveys* Vol. 21, No. 1 (1989).
29. F. Hayes-Roth, D.A Waterman and D.B. Lenat (eds.), *Building Expert Systems* (Reading, Mass.: Addison-Wesley, 1983).
30. P.M. Hill and J.W. Lloyd, The Gödel Report, *Technical Report TR–91–02* (Bristol: Computer Science Department, University of Bristol, 1991, revised 1992). See also [31].
31. P.M. Hill and J.W. Lloyd, *The Gödel Programming Language* (Cambridge, Mass.: MIT Press, 1994).

32. L.J. Holtzblatt, Diagnosing multiple failures using knowledge of component states, *IEEE Proceedings of AI applications* (1988).
33. N. Johns, *LPA MacPROLOG™ Graphics Manual* (London: Logic Programming Associates Ltd., 1991).
34. S.C. Kleene, *Mathematical Logic* (New York: John Wiley, 1967).
35. R.A. Kowalski, Problems and Promises of Computational Logic, in: J.W. Lloyd, *Computational Logic* (Berlin: Springer-Verlag, 1990) 1–36.
36. J.W. Lloyd, *Foundations of Logic Programming* (Berlin: Springer-Verlag, 1984).
37. J. McDermott, Preliminary Steps Toward a Taxonomy of Problem-Solving Methods, in: S. Marcus (ed.), *Automating Knowledge Acquisition for Expert Systems* (Boston: Kluwer Academic, 1988) 225–256.
38. J. Olsson, *An Architecture for Diagnostic Reasoning Based on Causal Models*, Ph.D. Thesis, Stockholm University/The Royal Institute of Technology, Stockholm, 1991.
39. D. Poole, Normality and Faults in Logic-Based Diagnosis, *Proc. Eleventh International Joint Conf. on Artificial Intelligence*, Detroit (1989) 1304–1310.
40. D. Poole, R.G. Goebel and R. Aleliunas, Theorist: a logical reasoning system for defaults and diagnosis, in: N. Cercone and G. McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge* (New York: Springer-Verlag, 1987) 331–352.
41. R. Reiter, A theory of diagnosis from first-principles, *Artificial Intelligence* 32 (1987) 57–96.
42. J.A. Robinson, A Machine-oriented Logic Based on the Resolution Principle, *Communications of the ACM* 12 (1965) 23–41.
43. M.J. Sergot, A Query-the-User Facility for Logic Programming, in: P. Degano and E. Sandewall (eds.), *Integrated Interactive Computer Systems* (Amsterdam: North-Holland, 1983) 27–41.
44. L. Steels, Components of Expertise, *AI Magazine*, Summer (1990), 29–49.
45. P. Struss, Model-Based Diagnosis of Technical Systems, Tutorial at *9th European Conf. on Artificial Intelligence*, Stockholm (1990).
46. P. Struss and O. Dressler, Physical Negation—Integrating Fault Models into the General Diagnostic Engine, *Proc. Eleventh International Joint Conf. on Artificial Intelligence*, Detroit (1989) 1318–1323.
47. A. Tarski, The semantic conception of truth, *Philosophy and Phenomenological Research* 4 (1944). See also [48].
48. A. Tarski, The Concept of Truth in Formalized Languages, *Logic, Semantics, Metamathematics* (Oxford: Clarendon Press, 1956).
49. W. Van Melle, A domain-independent production-rule system for consultation programs, *Proc. Sixth International Joint Conf. on Artificial Intelligence*, Tokyo (1979) 923–925.
50. R.W. Weyhrauch, Prolegomena to a Theory of Mechanized Formal Reasoning, *Artificial Intelligence* 13 (1980) 133–170.