Implementing Problem Solving Methods in CYC

Dimitrios Sklavakis

MSc in Artificial Intelligence
Department of Artificial Intelligence
University of Edinburgh
1998

Abstract

Although the CYC system is very good in answering questions through backtracking in its large knowledge base (KB), it can be illustrated in the case of systematic fault diagnosis that this is not enough to solve more complex problems that demand dynamic collection of information and KB updating. A solution proposed in this thesis is to provide CYC with a richer set of inference mechanisms and with the means to combine them into more complex problem-solving procedures.

The KADS methodology provides an excellent source both for inference mechanisms, its *inference types*, and ways to combine them, through its *inference structures* and *Generic Task Models*. Therefore, KADS provides the means to develop a rich set of inference mechanisms which will improve the reasoning power of CYC. In addition, KADS provides a systematic, task-oriented approach to knowledge acquisition which naturally complements the task-independent development of CYC's *Upper Ontology*. The combination of these two approaches provides the means to fill in the knowledge gap between these two extremes of the Ontology pyramid; the task-specific knowledge, which forms the base, and the Upper Ontology, which forms the top. A relevant issue discussed, is the problem of "brittleness" of Expert Systems, an issue fundamental for the creation and use of CYC.

The actual implementation of the Systematic Diagnosis problem solving method for faults in PCs and Automobiles, described in this thesis, demonstrates a way to implement KADS problem solving methods in CYC. The implementation maps directly the various layers of KADS *Expertise model* onto CYC's KB and its LISP-like programming language, SubL. The implemented system thus combines the declarative richness, transparency and expressiveness of the CYC KB with the conceptual analysis and structured search represented in the problem solving method.

Acknowledgements

The research reported here was carried out in the Department of Artificial Intelligence (DAI) with the cooperation of the Artificial Intelligence Applications Institute (AIAI). I would like first of all to thank my supervisors, Dr Helen Pain, from DAI, and Dr Stuart Aitken, from AIAI , for their patient and careful guidance during the course of the research reported here. Finally, I thank the Greek Ministry of National Economy, Directorate of International Economic Organizations, who provided financial support during my year of study via NATO Science Fellowship 107559/DOO 1094/25-7-97.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem

The principal goal of this thesis is to take a Problem Solving Method (PSM) from the Knowledge Based System (KBS) development methodology KADS ([Schreiber *et al.* 93], [Tansley & Hayball 93]) and implement its component inference steps in Doug Lenat's CYC KBS ([Lenat & Guha 90]).

Problem Solving Methods (PSMs) are conceptual models of problem solving which describe the elementary inference steps that must be performed for the solution of various tasks such as diagnosis, planning and design. They were developed in the context of methodologies for KBS design such as KADS. The task considered in this thesis is diagnosis and especially fault diagnosis in Personal Computers (PC). A small - but conceptually significant - extension of the task in the domain of Automobile fault diagnosis is also considered.

The implementation medium for the diagnosis PSM is CYC, a large Knowledge Base (KB) designed to contain enough knowledge to perform common sense reasoning. It is the product of a ten-year project, started in 1984 and ended in 1995, and is still under development ([Lenat 95]).The motive for building CYC was to overcome the "brittleness" of Expert Systems (ES), their inability to fall back on "first principles" when they encounter novel situations. The key problem was regarded to be analogical reasoning in all its forms. The overwhelming attempt to develop this kind of reasoning was divided in two sub-goals, as they are quoted from [Lenat & Guha 90]:

1. Breaking down the phenomenon, i.e. analogical reasoning, into its various subtypes and then handling each one.

2. Having a realistically large pool of (millions of) objects, substances, events, sets, ideas, relationships, etc., to which to analogize.

The development of CYC was directed mostly toward the second goal and much less to the first one. As a result, CYC now contains 10^6 common sense axioms expressed using a vocabulary of 10^5 concepts and just a handful of inferencing methods (backward and forward chaining, modus ponens, modus tollens, equality). The main method is backward chaining with resolution supported by special-purpose heuristic modules. The implementational details of CYC inferencing are discussed in the next chapter.

1.2 Motivation

The motive for implementing problem solving methods in CYC is to increase its inferencing power through the implementation of more inference types and methods.

Inference types, *inference structures* and *generic tasks*, another name for problem solving methods, are the conceptual products of research for the development of the second-generation expert systems. This research focused on distinguishing the knowledge-level ([Newell 82]) of an ES from its implementation level. These issues are very well illustrated in [Steels 90] and [Chandrasekaran 86].

The KADS methodology incorporates systematically all these concepts through its *Expertise Model*, divided into four layers, the *Domain, Inference, Task* and *Structure* layers, and its library of *Generic Task Models* which are domain-independent models of problem-solving behaviour. All these concepts are discussed in detail in the next chapter. Therefore, KADS provides two features in the same time:

- A complete methodology for developing ES independently from its implementation medium

- Domain-independent problem-solving methods (Generic Task Models) to be implemented

The domain independence of both CYC and KADS and the lack of any task-specific problem-solving methods in CYC are the motivating factors for this thesis. A secondary motive is that, although CYC was built to support the creation of ES, it does not provide any methodology either for the knowledge acquisition of the expert knowledge or for the common sense knowledge that underlies the expert knowledge. All these issues are discussed in the last chapter, under the light of the implementation of one problem-solving method from KADS for the task of PC fault diagnosis. This latter task was chosen since it is one of the most well studied and developed tasks.

1.3 Summary of Results

The main result of this thesis is that, the system developed enhances the problem-solving ability of CYC. A simple argument for this is that now CYC can solve a *whole range* of problems (those lending themselves to Systematic Diagnosis) that it could not solve before. This is because the specific problem-solving method demands the dynamic collection of information from the user, the appropriate update of the KB and the consequent, dynamic choice of the inferencing path. The system itself has features that are quite desirable from an ES, like transparency in its reasoning, ability of some justification, ease of maintenance and extension. The "brittleness" is still there, both in the domain and inference levels, but this was a rather expected feature. Overcoming "brittleness" is a promising direction of research rather than a nuisance. This is because this "brittleness" is due to the "distance" between the expert knowledge needed by the system and the restricted common sense knowledge base that the specific CYC system had[1]. This issue is also discussed in the last chapter.

In Chapter 2, KADS and CYC are described in the level of detail necessary to understand Chapters 3 and 4. Chapter 3 describes the Analysis phase, as defined by KADS, applied to the problem of PC fault diagnosis. This chapter also describes the specific problem-solving method, Systematic Diagnosis, together with how and why it was selected. Chapter 4 describes the design decisions and the actual implementation of the

[1]The CYC system used contained just a part of the *Upper Ontology* (see at `http://www.cyc.com/cyc-2-1/intro-public.html`).

problem solving method in CYC. Finally, Chapter 5 discuss the results of the implementation as well as further issues and research directions.

Chapter 2

The Component Technologies: KADS and CYC

2.1 Introduction

In this chapter the two main components of this thesis will be described, namely the KADS methodology and the CYC system. Of course, the description will cover only the parts of these two components that are important for the understanding of the thesis. A detailed description is beyond the scope of this thesis. For a more detailed description the reader must refer to [Wielinga *et al.* 92] and [Tansley & Hayball 93] for KADS and to [Lenat & Guha 90] and `http://www.cycorp.com`, the WWW cite of CYCORP, for CYC.

2.2 The KADS Methodology

The KADS methodology is the result of the corresponding European research project (ESPRIT-I P1098)[1]. It does not seem that a single interpretation of the KADS acronym exists. Two possible interpretations are "**K**nowledge **A**cquisition and **D**omain **S**tructuring" and "**K**nowledge-based systems **A**nalysis and **D**esign **S**upport". However, the KADS acronym stands as a name by itself. It is a comprehensive methodology for

[1] The main reference is [Hesketh & Barett 90]. It is available from: KADS Information, Touche Ross Management Consultants, Peterborough Court, 133 Fleet Street, London, EC4A 2TR, UK

the development of KBS. The initiative for the development of KADS was the lack of any methodology for the development of KBS. Another reason was the awareness from organisations using KBS that the development of this kind of system had a lot in common with the development of other information systems. Therefore, KADS is a complete method for the *Analysis* and *Design* of an information system which may be a KBS by its own or containing a KBS as one of its parts. This fact is reflected in both its Analysis and Design phases and the stages they contain:

1. Analysis phase
   - Process Analysis
   - Cooperation Analysis
   - **Expertise Analysis**
   - Constraints Analysis
   - System Overview
2. Design phase
   - Global Design
   - KBS Design

In the Analysis phase, the Expertise Analysis stage is the one concerned with the KBS part of the information system under development and it is the one that is of interest in this thesis. On the other hand, the Design phase will not be needed since it is not used; the pre-selection of the implementation medium, i.e., the CYC KBS, restricts and guides the design decisions that have to be made. A good overview of KADS can be found in [Wielinga *et al.* 92]. For a detailed description of the KADS methodology the reader may refer to [Tansley & Hayball 93].

2.2.1 Expertise Analysis

Before describing the Expertise Analysis stage of the Analysis phase, a major characteristic of KADS, which distinguishes it from the usual methods for developing KBS, must be noted.

The main activity when building a KBS is that of *Knowledge Acquisition*. Traditionally this process was viewed as a process of extracting knowledge from an expert, using various methods, and transferring (cf. encoding) this knowledge into the KBS. In KADS, a different view is adopted, regarding knowledge acquisition as a *modelling activity*. The KBS is not regarded as a container filled with knowledge but rather as a computational implementation of a desired behaviour. This behaviour is described in terms of *models*. Each model describes a particular aspect of the overall behaviour of the KBS, emphasizing certain characteristics and abstracting from others. Under this view, every stage of Analysis and Design in KADS produces a corresponding model of the overall KBS. Therefore, the result of Expertise analysis is the *Expertise model*.

The Expertise Model defines the desired problem solving behaviour (expertise) that the KBS must exhibit. It is the construction of this model that distinguishes KADS from other methodologies for information systems' development, such as Structured Analysis/Structured Design (Yourdon method) or Structured Systems Analysis and Design Method (SSADM). The construction of this model is based on two major assumptions:

1. The problem solving knowledge can be distinguished in *domain knowledge* and *control knowledge*. Furthermore, control knowledge can be distinguished in *inference*, *task* and *strategic* knowledge. These distinctions give four different layers of knowledge.

2. These four different layers have limited interaction between each other.

These four layers of the Expertise Model are described in more detail below.

The Domain Layer

In this layer a definition of the static domain knowledge is made, consisting of domain concepts, structures of concepts, attributes of concepts and relations between concepts. This knowledge is static in the sense that it describes some facts about the domain without specifying how this knowledge is going to be used. Therefore, this kind of description makes the knowledge implementation-independent to a certain degree, i.e., it may be used for different reasoning tasks such as diagnosis, teaching, explanation.

The structures of domain concepts and their relations are known in KADS as *Domain structures*. KADS does not provide a definitive and exhaustive formalism for what a Domain Structure should be as this decision depends strongly on the domain and the use of these structures. However, some general-purpose domain constructs are:

- **Concept**: the basic objects in the domain knowledge. It may correspond to either an individual or a collection. E.g., `component, system, subsystem`.
- **Attributes**: concepts may have attributes, e.g. `age(man)`, where `age` is an attribute of concept `man`.
- **Structure**: a complex object consisting of other concepts. E.g. `address`.
- **Set**: a collection of other domain constructs. All instances must be of the same type, i.e., concepts, structures, sets.
- **Relation**: they may be relations between concepts, e.g. `component isa subsystem` or relations between proper expressions, e.g., `temperature < 0 IMPLIES frozen (water)`.

The Inference Layer

This layer describes the basic inference capability of the KBS in terms of *inference types* and *inference roles*. It identifies which basic inferences are supported over the knowledge in the Domain Layer but it does not specify when or in what order these inferences actually happen.

Inference Types are primitive inference steps that can be performed on the domain knowledge. They are primitive since they are specified in terms of their input/output and their name which is a general description of what they do. For example, the *decompose* inference type takes a structured arrangement of objects and returns a collection of objects; the *select* inference type takes a collection (structured or unstructured) of objects and returns a filtered collection of objects. Therefore, inference types define ways the static domain knowledge may be used. See figure 2.1 for a detailed classification of inference types.

1. Concept Manipulation

   Generate Concept

   - **Instantiate**
   - **Generalise**
   - **Classify (Identify)**

   Change Concept

   - **Abstract**
   - **Specialise (Refine, Specify)**
   - **Assign_Value (Change_Value)**

   Distinguish Between Concepts

   - **Compare**
   - **Confirm**
   - **Select**

   Associate Concepts

   - **Match (Associate, Relate, Map)**

2. Structure Manipulation

   Build or Destroy Structure

   - **Assemble (Aggregate, Compose, Augment)**

   Re-arrange Structure

   - **Transform**
   - **Sort**
   - **Parse**

Figure 2.1: A hierarchy of Inference types

Domain roles define functions that domain structures may perform in various inference types. For example, in a PC fault diagnosis system, a specific component may be either a *hypothesis* to be selected by a *select* inference type or a *conclusion* to be made by a *confirm* inference type. These are two different roles for the same domain concept. Therefore, domain roles describe the static domain knowledge from a more problem-solving specific point of view. Domain roles may be classified according to the way they are used: they may be Input, Output and Intermediary (both Input and Output) roles.

Inference types and Domain roles are combined in *Inference Structures*. An Inference structure is a network of inference types and domain roles. In figure 2.2 an inference structure for fault diagnosis in an audio system is shown.

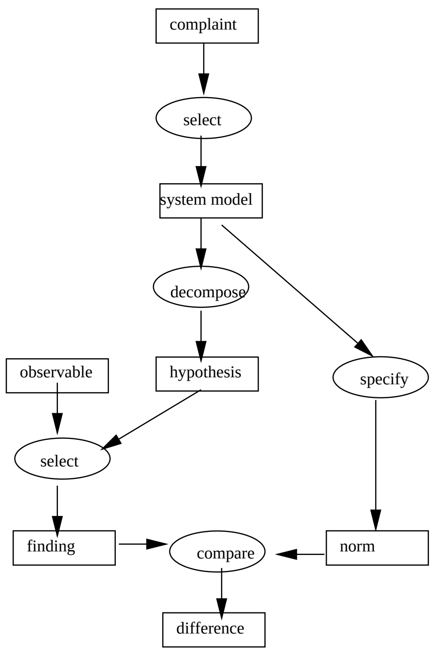

Figure 2.2: An Inference Structure for diagnosing faults in an audio system

The Task Layer

The Task Layer describes how the individual inferences described in the Inference layer may be sequenced in order to achieve each of the required problem-solving goals. The knowledge in this layer is defined in terms of *Task Structures*. These are usually written in pseudo-code which comprises simple sequences of inferences combined with some conventional control structures, such as conditionals (IF...THEN...ELSE), repetition (FOR, WHILE, REPEAT), or more complicated, such as pipelining and recursion. In figure 2.3 the Task Structure for performing systematic diagnosis is shown. Variables with a '+' are instantiated (input) while these with a '-' are uninstantiated (output).

```
Systematic Diagnosis(+complaint,+possible observables,-hypothesis) by
 select1(+complaint, -system model)
 REPEAT
  decompose(+system model, -hypothesis)
  WHILE number of hypotheses > 1
   select2(+possible observables, -variable value)
   select3(+hypothesis, -norm)
   compare(+variable value, +norm, -difference)
  system model <- current decomposition level of system model
 UNTIL confirm(+hypothesis), i.e. system model cannot be decomposed
       further
```

Figure 2.3: Task Structure for Systematic Diagnosis (pseudo-code)

The Strategy Layer

It is the final layer of the Expertise Model. It describes the knowledge for Task Structure selection, sequencing, planning or repairing (when a task fails). This layer will not be described since it is not used here[2].

[2] The main reference used in this thesis for KADS is [Tansley & Hayball 93], which describes a slightly modified version of KADS-I. A newer version, KADS-II or CommonKADS is now developed, but it is compatible with the parts of KADS described in this thesis. The only difference is that the Strategy layer is removed from the Expertise model, which is not important since this layer is not used here.

2.2.2 Generic Task Models (GTMs)

One of the goals for the KADS methodology was to overcome the *knowledge acquisition bottleneck*. This is the inherent difficulty in the process of extracting the problem-solving knowledge from an expert and encoding it in a computer system. Although the construction of the Expertise Model as it has been described is a step forward, it still remains a significant effort to construct the Expertise Model from scratch. For this reason KADS provides a library of Generic Task Models. These are Expertise Models (tested and verified) but without a Domain Layer. This relationship is shown in figure 2.4

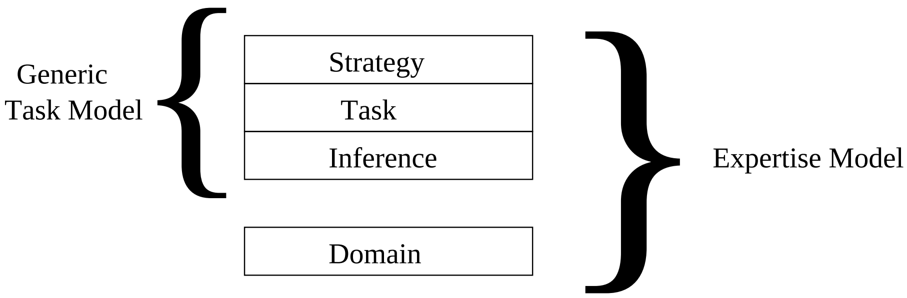

Figure 2.4: Relationship of Generic Task Model to Expertise Model

With GTMs provided, the construction of the Expertise model consists of three activities:

- Analyse Static Knowledge: this is the construction of the Domain Layer.
- Select Initial GTM: the selection of an initial GTM that will guide the construction of the Expertise Model. This selection is guided by the classification of the GTM library into a hierarchy of GTMs according to the specific problem-solving tasks that the KBS has to perform. This hierarchy is shown in figure 2.5. The GTM defines the Inference, Task and Strategy Layer. It also assists in further knowledge acquisition through the description of Domain roles that are needed for the various inference types.
- Construct Expertise Model: completion of the Expertise model by filling in the details of the three upper layers and putting them together with the Domain layer.

1. SYSTEM ANALYSIS
   - Identification
     - Diagnosis
       * Single Model Diagnosis
         + Systematic Diagnosis
       * Multiple Model Diagnosis
         + Mixed Mode Diagnosis
     - Verification
     - Correlation
       * Assessment
     - Monitoring
     - Classification
       * Simple Classification
       * Heuristic Classification
       * Systematic Refinement
   - Prediction
     - Prediction of Behaviour
     - Prediction of Values
2. SYSTEM MODIFICATION
   - Repair
   - Remedy
   - Control
   - Maintenance
3. SYSTEM SYNTHESIS
   - Design
     - Hierarchical design
     - Incremental design
   - Configuration
     - Simple Configuration
     - Incremental Configuration
   - Planning
   - Scheduling
   - Modelling

Figure 2.5: A hierarchy of Generic Task Models in the library

2.3 The CYC System

CYC is the implementation system for this thesis. CYC is a very large, multi-contextual knowledge base and inference engine. The project for its development started by the Microelectronics and Computer Technology Corporation (MCC) in the early 1984 and ended in 1995. In that year the CYCORP company was created, to work further on this project. The original idea behind CYC, introduced and developed by Doug Lenat, is that, to perform any kind of reasoning in a consistent and flexible way, any intelligent agent must have a considerable amount of common-sense, pre-scientific knowledge ([Lenat & Guha 90]). This kind of knowledge includes heuristics (rules of thumb) as well as assertions (facts) about the real world that can be known to a mechanical intelligence only if it is told about them. This "teaching" is actually implemented through manual knowledge editing. Using this considerable amount of knowledge - estimated to be at about 10,000,000 rules and facts - the agent can then perform common-sense reasoning and furthermore expert-like reasoning. The CYC system includes the following components[3]:

- The CYC Knowledge Base (KB)
- The CycL Representation Language (CycL)
- The CYC Inference Engine
- Interface Tools

2.3.1 The CYC Knowledge Base (KB)

The CYC knowledge base (KB) consists of:

- *constants*, also called *terms* or *units*. Constants form the basic vocabulary of the KB.
- *assertions* about these constants, which include *facts* and *rules*. All the assertions are formally expressed in a representation language, CycL, described below.

[3]In the following sections the version of CYC in the Artificial Intelligence Applications Institute (AIAI) is described and only to the extent necessary to understand the thesis.

Each CYC constant is the representation of a concept. In CycL, by convention, the names of constants begin with the prefix '#$' (read "hash dollar"), e.g. #$PCComponent[4]. A constant can represent a *collection* (such as the collection of PC components, #$PCComponent), an individual object (such as a particular PC component, #$VideoCard), a relation (a predicate, function, e.g., #$functionalPartOf) and so on.

All CYC KB constants form a hierarchy of *Collections*, subsets of them and instances of them. These hierarchical relationships are expressed with two special predicates, #$genls (meaning "subset of") and #$isa (meaning "element of"). This hierarchy is very important, it forms the CYC *Ontology*, and therefore must be described in more detail.

The CYC Ontology

CYC constants can either denote sets, like "the set of all PC components", or individuals, like "the video card". Every term in CYC is an element of #$Thing, the universal collection. #$Thing is partitioned into #$Individual and #$SetOrCollection.

#$Individual denotes the set of all things which are not sets. Individuals in the CYC KB include constants such as #$VideoCard and #$Decompose (inference type).

#$SetOrCollection is partitioned into #$Set-Mathematical and #$Collection. In this thesis we will not use mathematical sets and therefore their properties as far as it concerns CYC are not described. In fact, collections are more important in the CYC ontology and used more often[5]. The important thing to know about collections is that they can have elements. Therefore, they can enter in set-theoretic relations like "element", "subset" and "superset".

Membership in a collection is typically expressed as "instance of" or "element of" or "is a", as in "Monitor is an instance of the collection #$PCComponent," or "Monitor is an element of #$PCComponent," or "Monitor is a #$PCComponent." If the terms

[4]All terms in the examples come from the problem domain of PC fault diagnosis as this is analysed in the next chapter. If the meaning of any of the terms prevents the understatnding of the examples, the reader can refer to its definition in the next chapter.

[5]The difference between mathematical sets and collections is that the former are defined *extensionally*, i.e. by their members, while the latter are defined *intensionally*, i.e. by their criteria for membership. Therefore, two sets with the same members are equal, while two collections may be still different.

"subset" and "superset" are used with reference to collections, they typically are intended to mean "more specific collection" and "more general collection", respectively.

The predicate #$genls is used to indicate that one collection is more general than another, that it is a "superset". For example,

```
(#$genls #$PCComponent #$PCSubSystem)
```

indicates that the collection of PC subsystems is a superset of the collection of PC components.

The predicate #$isa is used to indicate that a thing is an instance of (element of) a collection, as in

```
(#$isa #$VideoSystem #$PCSubSystem)
(#$isa #$VideoCard  #$PCComponent)
```

where #$VideoSystem is stated to be an instance of the collection of PC subsystems, and #$VideoCard as an instance of the collection of PC components.

The predicates #$isa and #$genls are strongly supported by the CYC system code. There are special datastructures and special code routines inside CYC which allow rapid, efficient reasoning about collection-membership and collection-supersethood using #$isa and #$genls.

To summarise, every CYC constant is an element of at least one collection. In fact, everything that can appear in a CYC expression is an element of some collection. Every collection, with the exception of #$Thing, is a subset of at least one other collection. These "subset" and "instance of" relations, expressed with assertions using #$genls and #$isa, make up the basic framework (ontology) of the CYC KB. Figure 2.6 shows some constants and the #$genls and #$isa relations between them. In this diagram, the following conventions are introduced:

- CYC constants are represented as text in rectangular boxes.
- A constant with bold text denotes a collection.
- A constant with normal text denotes an individual.
- Assertions involving binary predicates are shown as lines between constants.

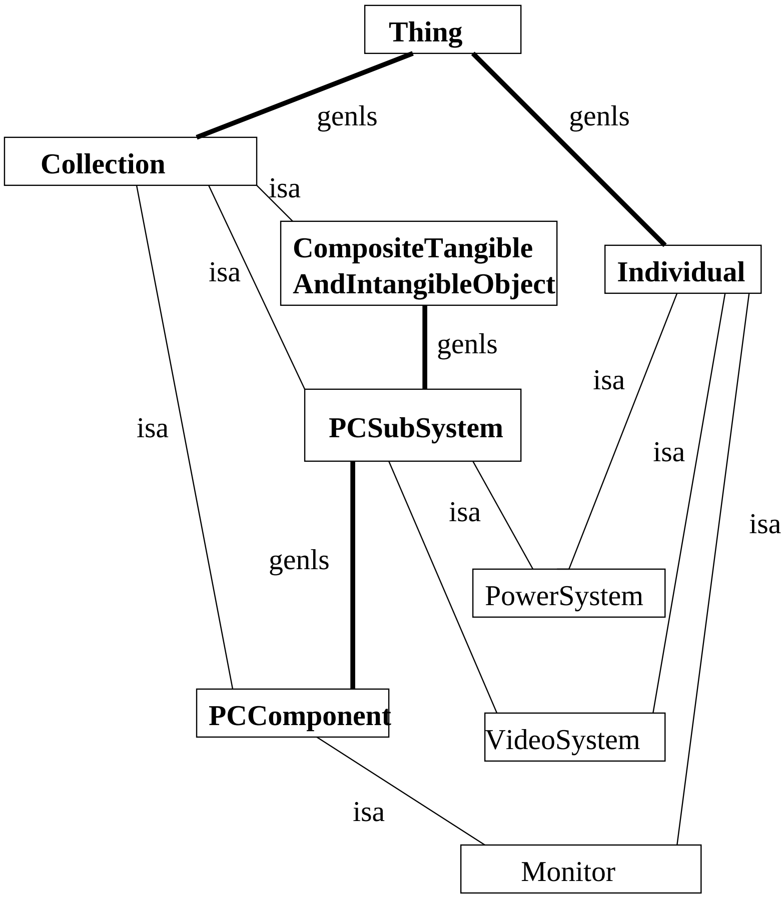

Figure 2.6: An Example of genls and isa

- #$isa assertions are shown as thin lines.
- #$genls assertions are shown as thick lines.
- More general collections are placed higher on the page than their subsets.
- Collections are placed higher on the page than their instances.

The diagram therefore indicates the following assertions:

```
(#$isa #$CompositeTangibleAndIntangibleObject #$Collection)
(#$isa #$PCSubSystem #$Collection)
(#$genls #$PCSubSystem #$CompositeTangibleAndIntangibleObject)
(#$isa #$PCComponent #$Collection)
(#$genls #$PCComponent #$PCSubSystem)
(#$isa #$PowerSystem #$PCSubSystem)
(#$isa #$PowerSystem #$Individual)
(#$isa #$VideoSystem #$PCSubSystem)
(#$isa #$VideoSystem #$Individual)
(#$isa #$Monitor #$PCComponent)
(#$isa #$Monitor Individual)
```

It is important to notice that, even though not directly indicated by the diagram, the following assertions also hold:

```
(#$isa #$Monitor #$PCSubSystem)
(#$isa #$VideoSystem #$CompositeTangibleAndIntangibleObject)
(#$genls #$PCComponent #$CompositeTangibleAndIntangibleObject)
```

These assertions are implied by two elementary properties:

- If B is a subset of A and X is an element of B, then X is an element of A too. In CYC terms: (#$genls B A) and (#$isa X B) implies (#$isa X A).
- If B is a subset of A and C is a subset of B, then C is a subset of A too. In CYC terms: (#$genls B A) and (#$genls C B) implies (#$genls C A).

As mentioned before, CYC has special inference mechanisms for inferring these kind of relationships.

2.3.2 The CycL Representation Language

CycL is a formal language whose syntax derives from first-order predicate calculus (the language of formal logic) and from Lisp. The vocabulary of CycL consists of *terms*. The set of terms can be divided into *constants*, *non-atomic terms* (NATs), *variables*, and a few other types of objects. Terms are combined into meaningful CycL *expressions*, which are used to make *assertions* in the CYC knowledge base (KB).

Constants

The CycL constants are the same as the KB constants described earlier. They make up the "vocabulary" of CycL. It must be remembered that they (usually) begin with the prefix '#$' (read "hash-dollar"). These characters may be omitted by certain interface tools, e.g., in the KE text interface described below. Some important naming conventions are:

- All CYC predicate names must begin with a lowercase character.
- All non-predicate constant names must begin with an uppercase character.

Variables

They are the common variables of any language. A variable may appear (nearly) anywhere a constant can appear. This gives to CycL some flavour of higher-order predicate calculus but this is not of interest in this thesis. Variable names must begin with a question mark and are ordinarily written in capital leters, e.g., ?TEST. Variables in CycL expressions can be either free or quantified. CycL provides the two main quantifiers of first-order predicate calculus; the universal quantifier #$forAll, and the existential quantifier #$thereExists. For expressiveness, it also provides three more existential quantifiers: #$thereExistAtLeast, #$thereExistAtMost, and #$thereExistExactly. Free variables are regarded to be universally quantified.

Formulas

CycL formulas combine terms into meaningful expressions. Every formula has the structure of a Lisp list: it is enclosed in parentheses, and consists of a list of objects, the arguments. The first argument may be a predicate, a logical connective, or a quantifier. The remaining arguments may be atomic constants, non-atomic terms, variables, numbers, strings delimited by double quotes ("), or other formulas. The simplest kind of formula is an *atomic formula*, a formula in which the first argument is a predicate, and all the other argument are terms:

```
(#$functionalPartOf #$VideoSystem #$Monitor)
(#$isa #$PowerSocket #$PCComponent)
(#$testAfter ?SUBSYSTEM1 ?SUBSYSTEM2)
```

The first two of the atomic formulas above are *ground atomic formulas* (GAFs), since none of the terms are variables.

Predicates

Every CycL atomic formula must begin with a predicate in order to be well-formed. The number of arguments a predicate takes is determined by its arity. A predicate is described as unary, binary, ternary, quaternary, or quinary, according to whether it takes 1, 2, 3, 4, or 5 arguments. Currently, no CycL predicate takes more than 5 arguments.

The type of each argument must be specified in the definition of the predicate, using the predicates #$arg1Isa, #$arg2Isa, etc. For example, suppose the predicate #$resultOfTest is defined by the following:

```
(#$isa #$resultOfTest #$BinaryPredicate)
(#$arg1Isa #$resultOfTest #$Test)
(#$arg2Isa #$resultOfTest #$PossibleObservableValue)
```

To be well-formed, every formula which has #$resultOfTest as its first argument must have a term which is an instance of #$Test as the second argument, and a term which is an instance of #$PossibleObservableValue as its third argument. So,

```
(#$resultOfTest #$Monitor #$BootTime)
```

is probably not well-formed. Though we can never be absolutely certain just from the names, #$BootTime could be an instance of #$PossibleObservableValue, but #$Monitor is probably not an instance of #$Test.

Logical Connectives

Complex formulas can be built up out of atomic formulas or other complex formulas by using logical connectives, which are special constants analogous to the logical operators of formal logic. The most important logical connectives in CycL are #$not, #$and, #$or, and #$implies. The three former have the obvious interpretation. The connective #$implies takes exactly two formulas as arguments. Like the "if-then" statement of formal logic, it returns true if and only if it is not the case that its first argument is true and its second argument is false. Here's an example:

```
(#$implies
    (#$and
     (#$diagnosisContext #$BootTime)
     (#$possibleHypotheses ?SUBSYSTEM)
     (#$testFirst ?SUBSYSTEM))
    (#$hypothesis ?SUBSYSTEM))
```

Assertions involving #$implies are very common in the CYC KB. We also call them *conditionals* or *rules*, and we often refer to the first argument as the *antecedent* and the second argument as the *consequent*. In the previous example, the antecedent is

```
(#$and
 (#$diagnosisContext #$BootTime)
 (#$possibleHypotheses ?SUBSYSTEM)
 (#$testFirst ?SUBSYSTEM))
```

and the consequent is

```
(#$hypothesis ?SUBSYSTEM))
```

Assertions

CycL formulas are used by Knowledge Editors (KEs) to enter assertions in the CYC KB and to ask questions to the KB. However, KB assertions are more than CycL formulas. They consist of many elements of which the most important are:

- a CycL formula: Formulas have been described in the previous section.

- a microtheory: Every assertion is contained in a single microtheory. A particular formula may be asserted into (or concluded in) more than one microtheory; when this is the case, there will be an assertion which has that formula in each of those microtheories. The largest number of assertions are currently in the #$BaseKB. All the assertions relative to the PC fault diagnosis are in the #$PCDiagnosisMt microtheory.

- a truth value: Attached to every assertion is a truth value that indicates its degree of truth. CycL contains five possible truth values, of which the most common are default true and monotonically true.

  By default, GAFs which begin with the predicates #$isa and #$genls are monotonically true, while all other assertions (including rules) are default true.

- a direction: Direction is a value associated with every assertion that determines when inferencing involving that assertion should be performed. There are three possible values for direction: *forward*, *backward*, and *code*. Inferencing involving assertions with direction forward is performed at assert time (that is, when a new assertion is added to the KB), while inferencing involving assertions with direction backward is postponed until a query occurs and that query allows backward inference. By default, GAFs have direction forward, while rules have direction backward. Changing the direction of rules to *forward* enables forward reasoning.

- a support: CYC uses a Truth Maintenance System (TMS) for its assertions. The support is the known support list of a TMS.

2.3.3 The CYC Inference Engine

The CYC inference engine handles modus ponens and modus tollens (contrapositive) inferencing, universal and existential quantification, and mathematical inferencing. It uses contexts called microtheories to optimize inferencing by restricting search domains. CYC also includes several special-purpose inferencing modules for handling a few specific classes of inference. One set of modules handles reasoning concerning collection membership, subsethood, and disjointness. Another handles equality reasoning.

Inferencing is initiated by an ASK operation. An ASK performed with direction *:forward* will simply do KB lookup; an ASK performed with direction *:backward* will initiate backward inferencing. Backward inferencing can be regarded as a search through a tree of nodes, where each node represents a CycL formula for which bindings are sought, and each link represents a transformation achieved by employing an axiom in the knowledge base.

2.3.4 Interface Tools

The Interface Tools that are of interest for this thesis are:

- The CYC Knowledge Base Browser
- The Knowledge Editing Text (KE Text) facilities
- The SubLanguage Interactor (SubL)
- The Functional Interface (FI-interface)

2.3.5 Interface Tools - The KB Browser

The CYC KB Browser is the main interface tool for accessing the CYC Knowledge Base (CYC KB). It provides a means for browsing the KB in a number of different ways, a means for querying the KB, and (for registered users) a means for modifying or adding to the KB itself. From the KB Browser, it is possible to reach virtually all other areas of the CYC System simply by following HTML links. Through the KB Browser the following operations may be performed:

- Creating, Viewing, Searching for and Editing Constants
- Adding, Viewing, Searching for and Editing Assertions

2.3.6 Interface Tools - The Knowledge Editing Text (KE Text) facilities

Introduction

KE Text is an ascii text format for specifying changes to a CYC KB. It uses a mixed "frame and formula" syntax and is batch-processed to add those changes to a CYC Server machine. KE Text (Knowledge Editing Text) is handled by two facilities: KE-File, which loads a file in KE Text format, and the Compose page in the CYC Web Interface.

KE Text Syntax

KE Text syntax is just a syntactic/notational variation of CycL. To some extent, it is a holdover from when CYC was a frame-based system and CycL was a frame-based language.

KE Text Syntax - Notation:

Variables

Variables occurring anywhere in a KE text (e.g., inside rule statements) must begin with a question mark (?).

Constants

Known constants (i.e., constants which CYC already knows to exist) may be written with a preceding '#$' (e.g., #$Monitor , #$Decompose), but this is in no case necessary and usually is not desirable. Accepted practice is to write KE text without #$ characters.

Strings

Strings referred to in KE text (such as entries on the #$comment predicate for a constant) must be delimited by double quotes (e.g., "This is a string."), as in Common Lisp and C.

Expressions

Expressions in KE Text syntax are analogous to expressions in a programming language such as Lisp or C. In KE Text syntax, each expression must end with a period (.), and the period must be outside of a comment or a string. The general form of an expression in KE Text syntax is as follows:

```
<directive>: <data-object-or-object-sequence>.
```

A *directive* may be a reserved word (analogous to reserved words in a programming language) or a predicate. Note that reserved word directive names are not case-sensitive. For example, "constant" is the same as "Constant".

KE Text Syntax - Reserved Words:

Constant

If the reserved word is "Constant", the data object following the colon delimiter must be the name of a CYC constant (e.g., PCSubSystem, or TestAction, or some other CYC constant). For example:

```
Constant: PCSubSystem.
Constant: TestAction.
```

If the data object following the colon delimiter is not already known (by CYC) to be a CYC constant, then this constant is created. The microtheory is set by default to be BaseKB. The only exception to this is if the microtheory has previously been set via the **Default Mt** directive, in which case the use of the Constant directive leaves the microtheory unchanged.

In Mt

If the reserved word is "In Mt", the data object following the colon delimiter must be a known (i.e., already existing) microtheory.

Example:

```
In Mt: PCDiagnosisMt.
```

When an expression beginning with an In Mt directive is evaluated, it causes the default entry microtheory to be set to the named microtheory. This setting persists until the next occurrence of an In Mt directive, Default Mt directive or a Constant directive.

Direction

The Direction directive sets the default direction for the assertion immediately following. It must be followed either by the constant *forward* or the constant *backward*. Note that, by default, ground atomic formulas have a "forward" direction and rules have "backward" direction. It is most commonly used to assert rules with "forward" direction.

F

If the reserved word is "F" (for "formula"), the data object following the colon delimiter must be a well-formed CycL Formula.

The constants referred to in the CycL formula must already be known to CYC (i.e., must already exist, perhaps as a result of being created at some previous point in the KE text).

Examples:

```
F: (implies
    (resultOfTest
        (TestFn PCSystem ConfirmSensorily ProblemContext) ?PROBLEM)
    (diagnosisContext ?PROBLEM)).


F: (possibleResultOfTest
    (TestFn PCSystem ConfirmSensorily ProblemContext) BootTime NotNormal).


F: (functionalPartOf VideoSystem PCSystem).
```

Default Mt

If the reserved word is "Default Mt", the data object following the colon delimiter must be a known (i.e., already existing) microtheory. For example:

```
Default Mt: PCDiagnosisMt.
```

When an expression beginning with a Default Mt directive is evaluated, it causes the default microtheory to be set to the named microtheory. This setting persists until the next occurrence of a Default Mt or In Mt directive, or the end of the file/text being processed. Note that this directive is stronger than the In Mt directive, since it prevents each occurrence of a Constant directive from resetting the default microtheory to BaseKB. This directive makes it easier to process all (or most) of the expressions in a file/text segment in the same microtheory.

Predicate Directives

The second type of directive comprises CYC predicates occurring within the scope of a (previously occurring) Constant directive. The Constant directive sets the "current" constant, which then is understood to be the first argument to assertions generated from the following predicate directive expressions. Note that predicate directive names, unlike reserved word directive names, are case-sensitive.

Each predicate directive is followed by a colon delimiter, one or more data objects, and a period. That is, the form of a predicate expression in KE Text syntax is

```
<predicate>: <data-object-1> [<data-object-2>...<data-object-n>].
```

The data objects following the colon delimiter comprise the additional argument(s) to the predicate in the predicate directive.

Example:

```
constant: PCSubSystem.
isa: Collection.
genls: CompositeTangibleAndIntangibleObject.
comment: ``The collection of all PC sub-systems, like the
#$VideoSystem, #$PowerSystem, #$KeyboardSystem.''.
```

In this example, the Constant directive sets the "current" constant to be PCSubSystem. PCSubSystem is then assumed to be the first argument to assertions formed from the three following predicate directive expressions (the expressions which begin with "isa", "genls", and "comment").

If the predicate directive is the name of a binary predicate (such as isa and comment), each of the data objects following the colon delimiter is assumed to be part of an assertion in which the predicate directive is the predicate, the default constant is the first argument, and the data object is the second argument. So, when evaluated and processed, the KE text fragment in the example above would result in the addition of the following three assertions to the KB:

```
(#$isa #$PCSubSystem #$Collection)
(#$genls #$PCSubSystem #$CompositeTangibleAndIntangibleObject)
(#$comment #$PCSubSystem ''The collection of all PC sub-systems,
 like the #$VideoSystem, #$PowerSystem, #$KeyboardSystem.'')
```

The same assertions could have been introduced using F: directives

Example:

```
F: (isa  PCSubSystem  Collection)
F: (genls  PCSubSystem  CompositeTangibleAndIntangibleObject)
F: (comment  PCSubSystem ''The collection of all PC sub-systems, like the
            #$VideoSystem, #$PowerSystem, #$KeyboardSystem, #$FloppySystem .'')
```

Note that this mechanism cannot be used for assertions involving unary predicates. For example, #$hypothesis is a such a predicate. Assertions using this predicate could be entered with an expression such as this:

```
F: (hypothesis PowerSystem) .
```

Comments in KE Text

Comments (text to be read by a human, but not interpreted or entered by a program) are allowed in KE text. The comment indicator is the semi-colon (;), as in Common Lisp. Lines beginning with a semi-colon will be ignored.

Order of Expressions

Expressions in KE text are evaluated and processed in the order of their occurrence in the text.

2.3.7 Interface Tools - The SubLanguage (SubL) Interactor

SubL is a computer language built by members of the CYC team. SubL was written to support the CYC application, allowing it to run both under Lisp environments and as a C application generated by a SubL-to-C translator.

SubL[6] is intended to be somewhat similar to Common Lisp, with features that are complex or rarely-used or difficult to implement in C excised. Also, unlike Common Lisp, SubL is not a purely functional language. Several Common Lisp constructs can only be used procedurally. In order to emphasize this difference, the following constructs have their names preceded either by 'c','p' or 'f' in SubL: `pif`, `pwhen`, `punless`, `pcond`, `pcase`, `csetq`, `cinc`, `cdec`, `cpush`, `cpushnew`, `cpop`, `clet`, `cmultiple-value-bind`, `cdo`, `cdotimes`, `cdolist`, `csome`, `cdohash`, `ccatch`, `cunwind-protect`, `cnot`, `cand`, `cor`, `fif`, `fwhen`, `funless`.

The SubL Interactor is an input window for evaluating SubL expressions.

2.3.8 Interface Tools - The Functional Interface (FI)

The CYC Functional Interface (FI) is an API (Application Program Interface) that external programs can use to query and update a version of CYC. The CYC FI provides the ability to find, create, kill, and rename constants, assert, unassert, ask, retrieve justifications for, and prove propositions, get and set application parameters, and a few other things. There is also a set of FI extensions for database integration. The commands of the functional interface can be invoked like normal lisp function calls from a lisp interactor such as the SubL Interactor of the CYC Web Interface as well as from other SubL functions. For a list of the FI functions used in this thesis see Appendix A.

2.4 Summary

In this chapter, a thorough description of KADS and CYC has been given. Specifically, KADS *Expertise model* was described with its four layers, namely the *Domain, Inference, Task and Strategy* layers, what they consist of, how they are built and what their inter-

[6]For more information see www.cyc.com/cyc-2-1/toc.html

relationships are. Also *Generic Task Models* were described, what they are and how they are used to build the Expertise model. All these parts of KADS are used in the next chapter for the Analysis phase of the problem solving method implementation.

For CYC, its *Ontology* was described, what this Ontology is, what it consists of and how it is represented through *CycL*, CYC's representation language. Also CYC's inference engine was described as well as the various interfaces provided for maintaining the KB (KB Browser), editing knowledge into the KB (Knowledge Editing Text) and executing Lisp code (the SubL Interactor and the Functional Interface). These parts of CYC will be used in chapter 4 to implement the Systematic Diagnosis problem solving method for PC and Automobile diagnosis.

Chapter 3

The Analysis Phase

3.1 Introduction

In the previous chapter (§2.2) it was noted that the analysis phase of KADS which is actually concerned with the development of the KBS is the *Expertise* analysis (§2.2.2) and that a library of Generic Task Models (GTMs) is provided. The building of the expertise model consists of three stages:

- the construction of the Domain Layer which contains the static knowledge,
- the selection of an appropriate initial GTM which contains a general description of the Inference, Task and (probably) the Strategic knowledge. More knowledge acquisition according to the *Domain roles* defined by the GTM and
- the filling of the details in the upper three layers and probably some modifications of the GTM.

In the following sections these three stages will be described in detail. But, before this, it must be made clear that the use of KADS as a methodology for analysing the system that will implement a problem solving method (PSM) is *completely independent* from the implementation medium which is CYC. This independence is twofold:

1. The problem solving method that will be selected should not necessarily come from KADS. It could have been developed in another context. For example, the Heuristic

Classification PSM ([Clancey 85]) was developed independently from the KADS methodology, although it can be found in the KADS GTM library (see Figure 2.5).

2. Even if the PSM was taken from KADS GTM library, still another approach for ES development could be used, e.g. rapid prototype development and further refinement. However, most of the PSM's power as part of KADS would have been lost, since there would be neither domain roles to guide the knowledge acquisition nor the Task structure to guide the PSM. If KADS was not used, one could try to implement the PSM in CYC by extending the ontology with new terms and rules using general knowledge acquisition techniques, and then using backward inference to implement the various steps of the PSM.

3.2 Knowledge Acquisition for the Domain Layer

There are two main sources of domain expert knowledge: bibliography and the human domain experts. The author has relatively good personal experience of PC troubleshooting, gained in a six month period of assembling and repairing PCs. Therefore, certain domain knowledge was at hand through self-introspection. However, more systematic knowledge was needed and it was found in the WWW site of the PCGuide magazine's Troubleshooting Expert (www.pcguide.com/ts/x/index.htm) developed by Charles M. Kozierok. This "Expert" is actually a set of menus containing questions about the PC system status and possible answers in the form of HTML links that guide the user to find any problem related to a PC. From the author's personal experience, and having studied the "Expert", it can be claimed that it is one of the most complete ever seen. The "Expert" has three main categories of troubleshooting *contexts*:

- Troubleshooting Boot Problems
- Troubleshooting The System Overall
- Troubleshooting Specific Components

The knowledge analysed is only that related to the first context, the boot-time problems troubleshooting. This may seem a major limitation but it is not, since the development of the KBS is only to test the feasibility and effectiveness of implementing problem solving methods in CYC; therefore, the KBS serves just as an experimental model and not as a fully functioning expert system per se.

After a thorough study of the “Expert”, the following domain structures appeared as significant in the diagnosis process:

1. A PC system model, a hierarchy of simple components and composite components, consisting of more simple components.

2. Testing knowledge, consisting of three other, more specific, domain structures:

    - Questions (Tests) that the “Expert” made about the PC system status,
    - Possible results of these Tests and
    - Actions taken according to each of these possible results.

It is quite obvious from the description of the emerging domain structures that they are far from being clearly defined. This is the *Knowledge Acquisition bottleneck* and the author was caught in it. Necessarily, the author used the help of KADS, its Generic Task Models and the *Domain roles* that these provide. Consequently, the second stage had to be entered, that is the selection of a GTM.

3.3 Selecting a Generic Task Model

One of the main problems when using KADS is the selection of a GTM. In fact, KADS provides little guidance for this selection and the decision comes back to the knowledge engineer. Criticism about this lack of guidance as well as about the potential dangers of selecting the wrong GTM can be found in [Rademakers & Vanwelkenhuysen 93].

The task for selecting a GTM in the case of PC fault diagnosis was quite simple; in figure 2.5 from [Tansley & Hayball 93], specific GTMs are given for the task of diagnosis. For simplicity reasons, and because of the way the “Expert” was designed, the Systematic Diagnosis GTM was selected. However, it is important to note that other GTMs were applicable, such as the Heuristic Classification GTM, a well established method for diagnosis, first developed by Clancey (see [Clancey 85]). The problem of more than one GTM applying to a specific task can be further explored in

[Rademakers & Vanwelkenhuysen 93]. The inference structures for the two GTMs are shown in figure 3.1.

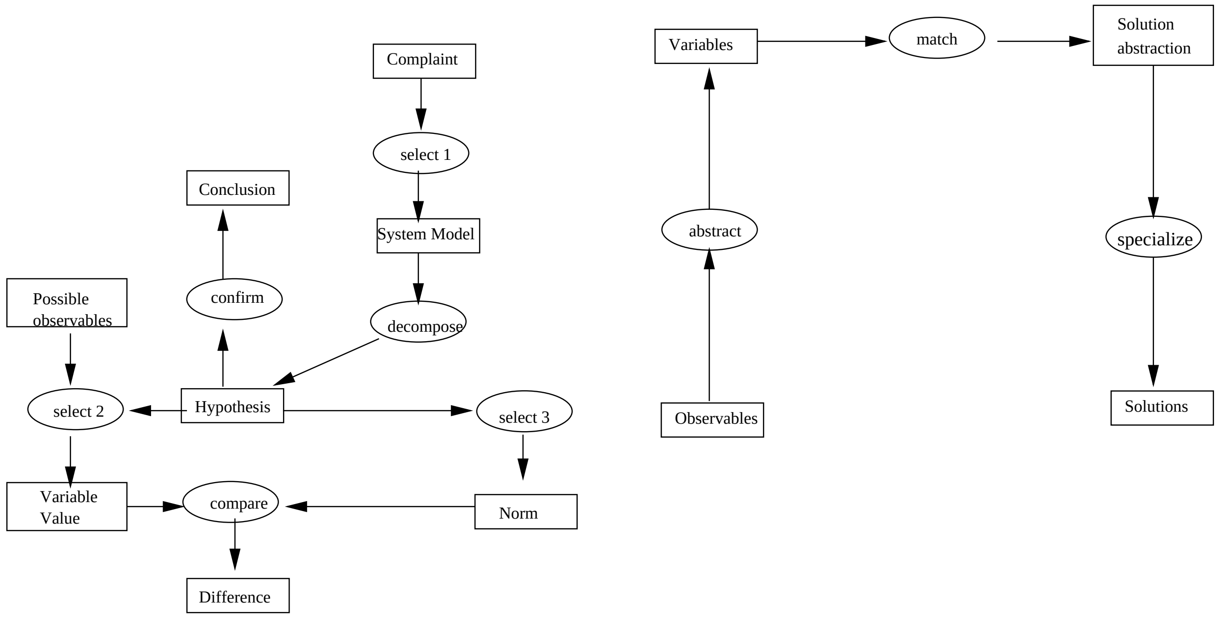

Figure 3.1: Inference structures for Systematic Diagnosis (left) and Heuristic Classification (right) GTMs

The selection of a GTM makes the construction of the Domain layer easier through the specification of *Domain roles* for the inference types included in the inference structure of the GTM. To illustrate this, each inference type and its associated domain roles of the Systematic Diagnosis GTM (Localisation version) are presented in table 3.1, as adapted from [Tansley & Hayball 93].

Note, that the Domain roles are customised for the Localisation version of the Systematic Diagnosis GTM. The actual, generic Domain Roles are given in table 3.2.

3.4 Domain Roles and Domain Layer

After having presented the Domain roles described by the GTM, it is much easier to construct the Domain layer of the KBS. The first domain structure that prevails the whole procedure is the *System model*. The Systematic Diagnosis GTM suggests a *consists-of*

Inference	**Input Role**	**Output Role**	**Method and Knowledge**
Select 1	Faulty system description	Consists-of model	Direct association - System behaviour and structure
Decompose	Consists-of model	(Sub)-System containing faulty component	Descending consists-of tree Consists-of structure
Select 2	(Sub)-System containing faulty component. Observable Output Values.	Observed Output Value	Generate and test- Test methods
Select 3	(Sub)-System containing faulty component	Expected Output	Direct association - System behaviour
Compare	Observed Output Value	Decision Class	Compares values Significance of differences
Confirm	(Sub)-System containing faulty component	Yes/No (+fault location)	Primitive part reached System structure

Table 3.1: The Systematic Diagnosis Inference Types and their Domain Roles

Domain Role	**Localisation**
Complaint	Faulty system description
System model	Consists-of model
Possible observables	Observable output variables
Hypothesis	(Sub)-System containing faulty component
Variable value	Observed output value
Norm	Expected output
Difference	Decision class
Conclusion	Yes/No (+ fault location)

Table 3.2: Domain Roles for Systematic Diagnosis and the Localisation equivalents

system model, however, as we will see below, there is a great variety of system models to choose from and, moreover, a consists-of model is not the best for diagnosis in a PC system. This significant variation from the proposed model does neither come as a surprise nor as something unusual. The domain structures proposed by a GTM are simply general guidelines and are not restricting for the knowledge engineer. They are another point of view for the domain knowledge, one more task-oriented. The same holds even for the inference structures of the GTMs as we will see later. This flexibility of the GTMs is discussed in [Rademakers & Vanwelkenhuysen 93] and [Wielinga *et al.* 92].

The prevalence of the PC system model and the inadequacy of a *consists-of* model are discussed below.

3.4.1 The System Model

The first thing that must be understood for the selection of a system model is the overall role that this plays in the Systematic Diagnosis problem solving method and especially in its Localisation version. The outline of the method is:

1. Begin with a general symptom of the system. Select a part of the system that probably contains the faulty component.

2. Decompose the suspected (sub)system into its *parts* which may be simple components or (sub)systems themselves.

3. Take the first/next component/subsystem and check if it is the one that contains the faulty component. If it does and it is a component then stop; if it is a (sub)system, go to 2. If it does not contain the faulty component, then repeat step 3 as far as there are candidate components/(sub)systems.

From the above outline it is clear that the system model should be a hierarchy of (sub)systems and their parts which in turn can be either simple components or further decomposable (sub)systems. But there are a lot of hierarchies. In [Steels 90] various such models are mentioned:

- Structural: these models describe the way the parts of the modelled system contain and form each other, like the model of a subway containing the stations, tracks and so on [Steels 90],

- Topological: these models describe the way the parts of the modelled system are connected to each other, like the model of a heating system containing the connecting pipes between the various parts of the system [Borst *et al.* 97],

- Functional: these models describe which parts are used by each (sub)system in order to carry out its function(s), like the model of a printed circuit board containing the *circuit operations* and the corresponding *hardware modules* that are involved in carrying out these operations [Vanwelkenhuysen 92].

The model selected for the Personal Computer (PC) fault diagnosis is a *functional* model. This decision is based in the fact that a PC is an information processing machine with central control and therefore exhibits the following two characteristics which distinguish it from other mechanical devices:

1. The *same* components take part in *different* functions. This results in *overlapping* (sub)systems and in different functional roles for the components, and therefore different behaviours and different possible faults, according to the function of the component.

2. Because of the central control of the CPU which imposes either predefined or dynamic sequences of component operations, the physical connections of the components do not always define the order of component operations and therefore the order of diagnosis.

The appropriateness of a functional model in the case of PCs is supported by its use in troubleshooting of electronic circuits in [Vanwelkenhuysen 92] and [Hamscher 88]. In contrast, the reader can refer to another, *mereotopological*[1] model, presented in [Borst *et al.* 97]. The two above characteristics mean that the functional model of the PC cannot be expressed in a static hierarchical structure but rather like a set of rules that describe different decompositions, according to which (sub)system is being decomposed and in which

[1] A mereotopological model is a combined structural and topological model

function it takes part. In addition, it must be noted that the decomposition process for the Systematic Diagnosis problem solving method requires specification of not only which (sub)parts of the (sub)system being decomposed are candidates for diagnosis but also *in which order* they will be diagnosed. An example will clarify the order's importance:

Example

Suppose that the PC does not produce any video signal on the monitor when it is turned on. This suggests that the video (sub)system of the PC has a fault. This (sub)system contains the following components: the motherboard, the video card and the monitor. It is obvious that the monitor should be checked first, otherwise no test can be performed on either of the other two components due to lack of feedback (control) from the (possibly) faulty monitor.

To summarise about the desired properties of the PC system model:

- It must provide a different decomposition of the PC (sub)systems in different cases,

- It must provide the order of testing for the components of the decomposed (sub)system and

- It must distinguish between decomposable (sub)systems and simple components.

All these properties lead to the following domain structures[2]:

CONCEPTS

[2]There are many more instances of PCComponent and PCSubSystem to be defined as concepts. A complete list can be found in the KE-text for the CYC KB in Appendix B.

PCComponent	A simple, non-decomposable PC component. A PCComponent is the lowest level of the system's analysis.
PCSubSystem	A decomposable PC (sub)system. A PCSubSystem involves one or more PCComponents and/or PCSubSystems and it is an intermediate level of the system's analysis.
PowerSystem	The power system of the PC.
PowerSupply	The power supply device of the PC.

ATTRIBUTES

hypothesis(PCSubSystem): The PCSubSystem is the current candidate for diagnosis.
possibleHypotheses(PCSubSystem): The PCSubSystem is one of the next candidates for diagnosis.
testFirst(PCSubSystem): The PCSubSystem is the first to be tested from all the other candidate PC subsystems; it is the next hypothesis.

RELATIONS

subsetOf(PCComponent, PCSubSystem): A PC component is the simplest PC subsystem
isa(PowerSystem, PCSubSystem): PowerSystem is a PCSubSystem.
isa(PowerSupply, PCComponent): PowerSupply is a PCComponent.

functionalPartOf(PCSubSystem-Whole,PCSubSystem-Part): The PCSubSystem-Whole involves in its function the PCSubSystem-Part.
testAfter(PCSubSystem-1, PCSubSystem-2): The PCSubSystem-2 must be tested immediately after the PCSubSystem-1.

The decomposition rules are missing. This is because their 'if' part - the antecedent - requires concepts not yet defined. These concepts are the subject of the next section.

3.4.2 The Testing Knowledge

The basic tool in the Systematic Diagnosis problem-solving method (PSM), as well as in any other diagnostic PSM, for carrying out the diagnostic procedure, is various Tests that must be done to provide information (knowledge) about the state of the system. This knowledge may concern the actual behaviour of the system's components, e.g. the absence of electric power or control information produced by the system, e.g., beep codes or screen messages. Conceptually, a Test is a question that the user must make to the system under diagnosis to extract knowledge about it. A Test is a structure consisting of three other concepts:

1. The PC subsystem to which it is related, i.e., to which the question is addressed,

2. The Action that the human user must make to perform the Test and

3. The Possible Observable (system variable) that the Test is asking about.

Although not part of a Test structure, there is a fourth concept related to it, the Possible Observable Value (system variable value) which is the result (answer) of the Test (question). A final concept, related to the Test's result, is the Result Type which describes what the result of a Test means for the diagnosis procedure, that is, what further decision it entails. All these lead to the following domain structures[3]:

CONCEPTS

[3]There are many more instances of TestAction, PossibleObservable, PossibleObservableValue and ResultType to be defined as concepts. A complete list can be found in the KE-text for the CYC KB in Appendix B.

TestAction	A physical action that the human user must make to perform a Test.
PossibleObservable	A system variable the values of which give information about the system status.
PossibleObservableValue	A possible value of a system variable.
ResultType	The type of a Test's result. These types are characterised from the kind of conclusions they lead relative to the PCSubSystem being currently diagnosed (hypothesis(PCSubSystem)). E.g., such a type can be Normal which denotes that the PCSubSystem currently being diagnosed is not faulty and therefore must be discarded as a hypothesis and a new hypothesis must be selected.
ConfirmSensorially	The action of confirming the existence of a PossibleObservable only by one's senses, e.g., visually, acoustically.
ElectricPower	The electric power that any PCSystem needs to operate.
Yes	Most of the Tests have as a possible result only 'Yes' or 'No'.
NotNormal	This type of result indicates that the result is not normally expected when the PCSubSystem related with it is working properly. Such a kind of result implies that the fault lies in the PCSubSystem which is the current hypothesis.

STRUCTURE

Test	**Constituent concepts**
	PCSubSystem
	TestAction
	PossibleObservable

ATTRIBUTES

possibleTest(Test): Test can be currently performed.

RELATIONS

isa(ConfirmSensorially, TestAction): ConfirmSensorially is a TestAction
isa(ElectricPower, PossibleObservable): ElectricPower is a PossibleObservable
isa(Yes, PossibleObservableValue): Yes is a PossibleObservableValue
isa(NotNormal, ResultType): NotNormal is a ResultType

possibleResultOfTest(Test, PossibleObservableValue, ResultType): The test Test has as a possible result the PossibleObservableValue which is of type ResultType.
resultOfTest(Test, PossibleObservableValue): The test Test gave as result the PossibleObservableValue when it was performed.

The rules introducing the possible Test are missing. This is because their 'if' part - the antecedent - requires a concept not yet define. This concept is the subject of the next section.

3.4.3 The Diagnosis Context

The Systematic Diagnosis GTM inference structure starts with a SELECT inference (select 1, see figure 3.1). A general symptom is entered by the user and an appropriate system model is chosen. This inference is slightly changed for PC diagnosis. What is actually asked of the user is to distinguish three major contexts of diagnosis:

1. Boot-time troubleshooting,
2. Run-time troubleshooting and
3. Component-specific troubleshooting.

This categorisation is significant since completely different rules are applicable in each context. This contextual dependency will be embedded in the antecedent part of the rules as an extra condition. The necessary domain structures are:

ATTRIBUTES

isa(BootTime, PossibleObservableValue)
isa(RunTime, PossibleObservableValue)
isa(ComponentSpecific, PossibleObservableValue)
diagnosisContext(PossibleObservableValue): The PossibleObservableValue is the current diagnosis context.

and the way they are used in rules is:

```
diagnosisContext(BootTime) and ...[more conditions]...  implies [consequent].
```

To satisfy all these specifications, the rules for decomposition of the PC system model will have the following general form:

```
diagnosisContext(PossibleObservableValue) and
hypothesis(PCSubSystem) and
resultOfTest(Test1, PossibleObservableValue1) and ...
....
and resultOfTest(TestN, PossibleObservableValueN)
IMPLIES
testFirst(PCSubSystem1) and
testAfter(PCSubSystem1, PCSubSystem2) and...
...
and testAfter(PCSubSystemN-1, PCSubSystemN).
```

while the rules for introducing new Tests and their corresponding results, will have the form:

```
diagnosisContext(PossibleObservableValue) and
hypothesis(PCSubSystem) and
```

```
resultOfTest(Test1, PossibleObservableValue1) and ...
.
.
.
resultOfTest(TestN, PossibleObservableValueN)
IMPLIES
possibleTest(Test) and
possibleResultOfTest(Test, PossibleObservableValue1, ResultType1) and ...
.
.
.
and possibleResultOfTest(Test, PossibleObservableValueN, ResultTypeN).
```

3.5 The Inference Layer

As described in the previous chapter (§2.2.1), the Inference layer defines the inference types that must be performed by the problem solving method. After the description of the domain layer in the previous section, the inference layer of the Systematic Diagnosis problem solving method for PCs is given in table 3.3, in terms of the various inference types as well as their input and output domain structures. Note that the input and output roles have now been substituted by actual domain structures, and that inferences *Select 2* and *Select 3* have been combined in one inference, *Select 2-3*. These changes are well situated into the development procedure of the Expertise model as described in [Wielinga *et al.* 92].

3.6 The Task Layer

The next layer is the Task layer, where the order of performing the inference types is described. This is done in terms of the pseudo-code given in figure 3.2. A comparison with the Task Structure given in figure 2.3 can show how this last one was modified.

The last layer of the Expertise model in KADS is the Strategy layer, however the

INFERENCE TYPE	**INPUT Role**	**OUTPUT Role**
Select 1	PC symptom's general nature (BootTime, RunTime, ComponentSpecific)	Problem solving Context
Select 2-3	Problem Solving Context Hypothesis Results of previous Tests	Possible Test(s) Possible Results of Test(s) Types of Possible Results
Decompose	Problem Solving Context Hypothesis Results of previous Tests	Hypothesis Possible Hypotheses
Compare	Last Test's Result Type	Next inference to be executed
Confirm	Last Test's Result Type	Faulty component

Table 3.3: The Inference Layer of the Systematic Diagnosis PSM for PCs

```
Systematic Diagnosis(+complaint,+possible observables,-hypothesis) by
 select1(+complaint nature, -diagnosis context)
 REPEAT
  decompose(+hypothesis, +diagnosis context, +previous tests' results,
            -hypothesis, -possible hypotheses)
  WHILE number of possible hypotheses > 1
   select2-3(+hypothesis, +diagnosis context, +previous tests' results,
             -possible test(s), -possible observables, -result types)
   compare(+last test's result type, -inference to be performed)
 UNTIL confirm(+hypothesis), i.e. system model cannot be decomposed further
```

Figure 3.2: The Task Structure for Systematic Diagnosis PSM for PCs

GTM for Systematic Diagnosis does not provide one and no such layer is needed for the implementation of this problem solving method.

3.7 Summary

In this chapter, the KADS Analysis phase was described as it was applied on the domain of PC fault diagnosis. Why the Systematic Diagnosis GTM was selected as the problem solving method and the guidance this provided for knowledge acquisition through the *domain roles* for its constituents inference types. What was the system model and the testing knowledge needed as domain knowledge for this GTM. Finally, the modifications that were made on some of its constituent inference types to fit the specific domain and the final Task structure. All these products will be used in the next chapter to implement the Systematic Diagnosis PSM in CYC. It is very important to remember that this phase is *completely independent* of the implementation medium, which in this thesis is CYC; however, it could be an expert system shell, CLIPS or PROLOG.

Chapter 4

The Implementation Phase in CYC

4.1 Introduction

The next step to implement the Systematic Diagnosis problem solving method (PSM) for PCs in CYC was to implement each one of the three layers of the Expertise model, namely the Domain, Inference and Task layers. Before presenting the implementational decisions, the main components of CYC, on which this implementation is based, are repeated:

- The CYC Knowledge Base (KB) consisting of terms structured in a Collection - Sub-collection - Instance hierarchy as well as rules connecting them,
- The CycL representation language which defines the KB terms as well as the rules between them,
- An 'Ask' interface for querying the KB, in a Prolog-like fashion, using the CycL language,
- The SubL language, a subset of Common Lisp,
- The Functional Interface (FI), a set of SubL functions that provide the means to perform all the necessary operations to the KB through SubL code and
- The SubL Interactor, an interface to execute SubL code.

All these CYC components are described in chapter 2.

4.2 The Overall Model

The common use of CYC is through its Ask interface. The user enters a CYC formula, possibly containing variables, and the CYC inference mechanism tries to find matchings for these variables through unification, using rules and facts in the KB. Under this scheme, solving a problem consists in defining the right knowledge, that is terms, facts and rules, making the right(?) questions and (probably) modifying the KB. This is a declarative way to solve problems. On the other hand, SubL provides a more procedural/functional way to do things. Through CYC's FI, it is possible to perform all the above operations from SubL code. Having all these in mind, the first and crucial problem to solve implementing a PSM is what goes where. In the case of this research, where the KADS method was used, this question is which expertise layer is implemented in which component of CYC. The final decisions that the author made in solving this problem were based in the following considerations:

1. The KADS methodology, through the Expertise model, provides a significant differentiation of problem-solving knowledge in *domain* and *control* knowledge. Since this distinction is at the heart of analysing problems, preserving it in the implementation phase keeps things simple and clear. Moreover, mixing domain and control knowledge was one of the major drawbacks of the early experts systems, like MYCIN ([Clancey & Letsinger 82]).

2. CYC's use is mainly through the declarative scheme described above. Although the goal of implementing PSM's in CYC is to make it more effective, the advantages of solving problems in a declarative way should not be lost in the obscurity of functional descriptions in Lisp (SubL) code. The overall guideline is that the implementation should permit the problem to be solved by the declarative way too. How this can be done will be explained later on in this chapter.

At the implementation level, these guidelines led to the following decisions:

1. The Domain layer was encoded in the CYC KB and

2. The Inference and Task layers were encoded in SubL code

This overall model can be seen in figure 4.1.

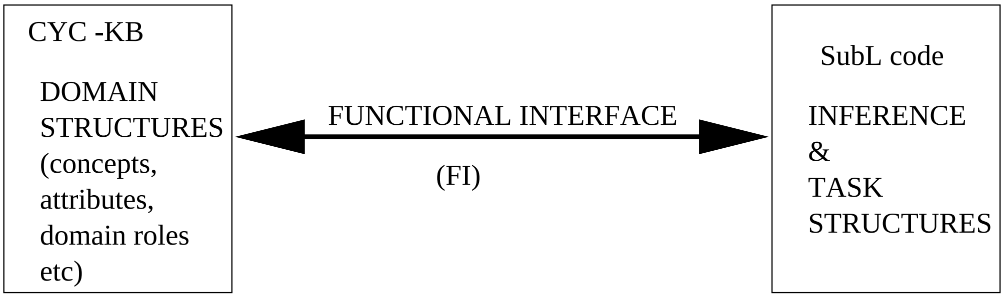

Figure 4.1: The Overall Model (schematic view)

4.3 The Implementation of The Domain Layer

Representing the Domain layer in the CYC KB is a significant task by itself. This is mainly for two reasons:

1. Every term in the CYC KB must be an instance of a Collection, except #$Thing, the top-most collection. Therefore, every domain structure defined in the analysis phase must be represented as the instance of another collection. Consequently, some domain structures must be fitted somewhere in the CYC *Ontology*[1]. The problem here lies in how to integrate the new knowledge that comes from a specific domain, which usually comprises very specialised concepts of the real world, with the knowledge already in CYC KB, which consists of very general concepts of the world. A *connecting, intermediate ontology* is needed to integrate the two kinds of knowledge in a realistic way.

2. Another dimension of the same problem occurs when different people are developping expert systems in CYC and consequently create new concepts and try to integrate them in the *already existing* CYC *Ontology*. This raises the issues of consistency, i.e. the same concepts may be defined as parts of contradictory parts of the Ontology, and duplication, i.e. different terms are defined that conceptually are identical. In [Lenat & Guha 90], these issues are discussed in detail.

[1]The AIAI version of CYC, on which this project was developed, contained only the *upper ontology* which can also be found at www.cyc.com/cyc-2-1/intro-public.html

The decisions required are quite hard and they primarily demand good knowledge of the given CYC ontology. However, such knowledge comes more by using than by studying the terms of the KB. The author studied the given ontology to the extent permissible in the time available. Therefore, it is not claimed that the solutions are either unique or the best. This is an issue discussed in the next chapter. However, the major points of these decisions are discussed below[2]:

4.3.1 The PC Diagnosis Microtheory.

In CYC, *microtheories* is a way of talking about a group of related assertions by representing this grouping explicitly with a Cyc unit ([Blair *et al.* 92]). The default microtheory is the #$BaseKB and any other microtheory is almost always a superset of this microtheory in a hierarchy which is constructed using the predicate #$genlMt. For all the reasons cited in [Blair *et al.* 92], all the domain knowledge for the Systematic Diagnosis PSM for PCs was grouped in a special microtheory, the *#$PCDiagnosisMt* (B:10).

4.3.2 The System Model.

The representation in CYC of the domain structures related to the system model knowledge was quite straightforward. Figure 4.2 illustrates the decisions about this representation. The meaning of all terms can be found in the corresponding part of the KE-text in Appendix B (B:21-207, B:691-717). Recall that in CYC, the #$isa predicate means "element of", while the #$genls predicate means "subset of". The important thing here is that the #$PCComponent collection is a sub-collection of the #$PCSubSystem collection. This is both for grouping every part of a PC in the latter and for distinguishing the simple, non-decomposable components by the former collection.

Another important aspect of the system model implementation is that, although a description of it is given in terms of the #$functionalPartOf assertions (B:691-717) and the general decomposistion rule[3] (B:681-690),

[2]Numbers in parentheses refer to number lines of the KE-text listing in Appendix B.

[3]Terms preceded by '?' are variables in CYC

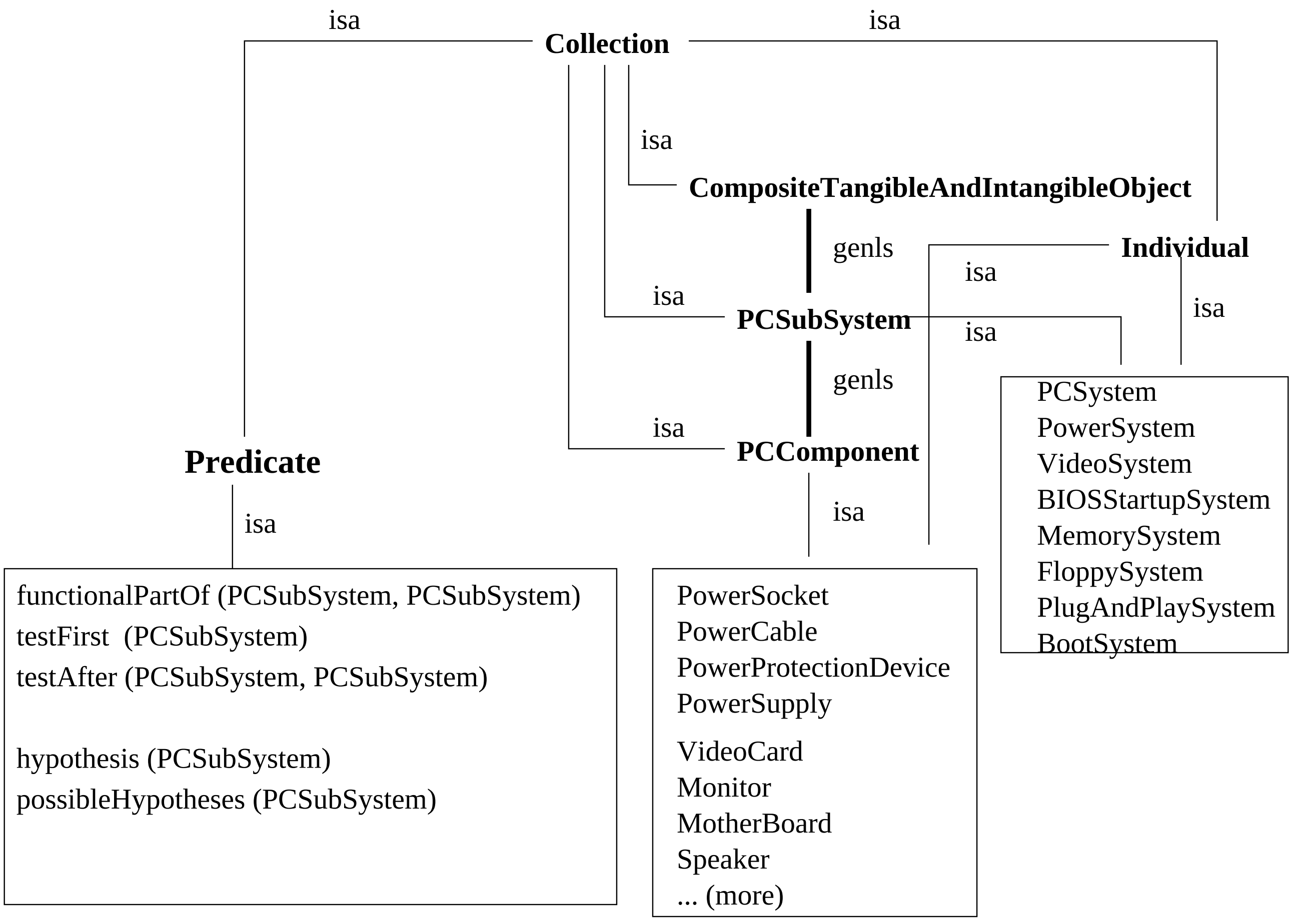

Figure 4.2: The System Model Hierarchy in CYC

```
(implies
 (and
  (diagnosisContext BootTime)
  (hypothesis ?HYPOTHESIS)
  (plausibleInference Decompose)
  (functionalPartOf ?HYPOTHESIS ?PART))
(possibleHypotheses ?PART)).
```

the order of testing is given through explicit decomposition rules which are context dependent, that is they have the general form:

```
(implies
 (and
  (diagnosisContext BootTime)
  (hypothesis HYPOTHESIS)
  (plausibleInference Decompose)
  (resultOfTest TEST_1 RESULT_1)
   ...
  (resultOfTest TEST_M RESULT_M))
 (and
  (testFirst PC_SUBSYSTEM_1)
  (testAfter PC_SUBSYSTEM_1 PC_SUBSYSTEM_2)
  ...
  (testAfter PC_SUBSYSTEM_N-1 PC_SUBSYSTEM_N))).
```

Example of Decomposition rule. The rule

```
(implies
 (and
  (diagnosisContext BootTime)
  (hypothesis VideoSystem)
  (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
  (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
  (plausibleInference Decompose))
```

```
(and
 (testFirst MotherBoard)
 (testAfter MotherBoard VideoCard))).
```

means that

IF

1. The problem solving context is Boot-time and
2. The video system is hypothesised as being in fault and
3. Something is written on the screen (VideoSignal=Yes) and
4. There is no video BIOS message and
5. A *Decompose* inference must be performed

THEN

1. Test first the Motherboard and
2. Test the Video card after the Motherboard.

It is the significance in the order of testing that imposes the encoding of explicit decomposition rules. *If the order of testing was not important, the explicit decomposition rules would not be necessary.* This would simplify the implementation of the system model in just the #$functionalPartOf assertions (B:691-717) and the general decomposistion rule(B:681-690). The explicit decomposition rules are grouped in the KE-text by component/subsystem (B:739-2130).

4.3.3 The Testing Knowledge

The representation of the domain structures related to the system testing knowledge was quite straightforward in CYC. Figure 4.3 illustrates the decisions about this representation. The meaning of all terms can be found in the corresponding part of the KE-text in Appendix B (B:214-559). There is a technicality concerning the representation of the Test structure. It is represented via a #$NonPredicateFunction, #$TestFn

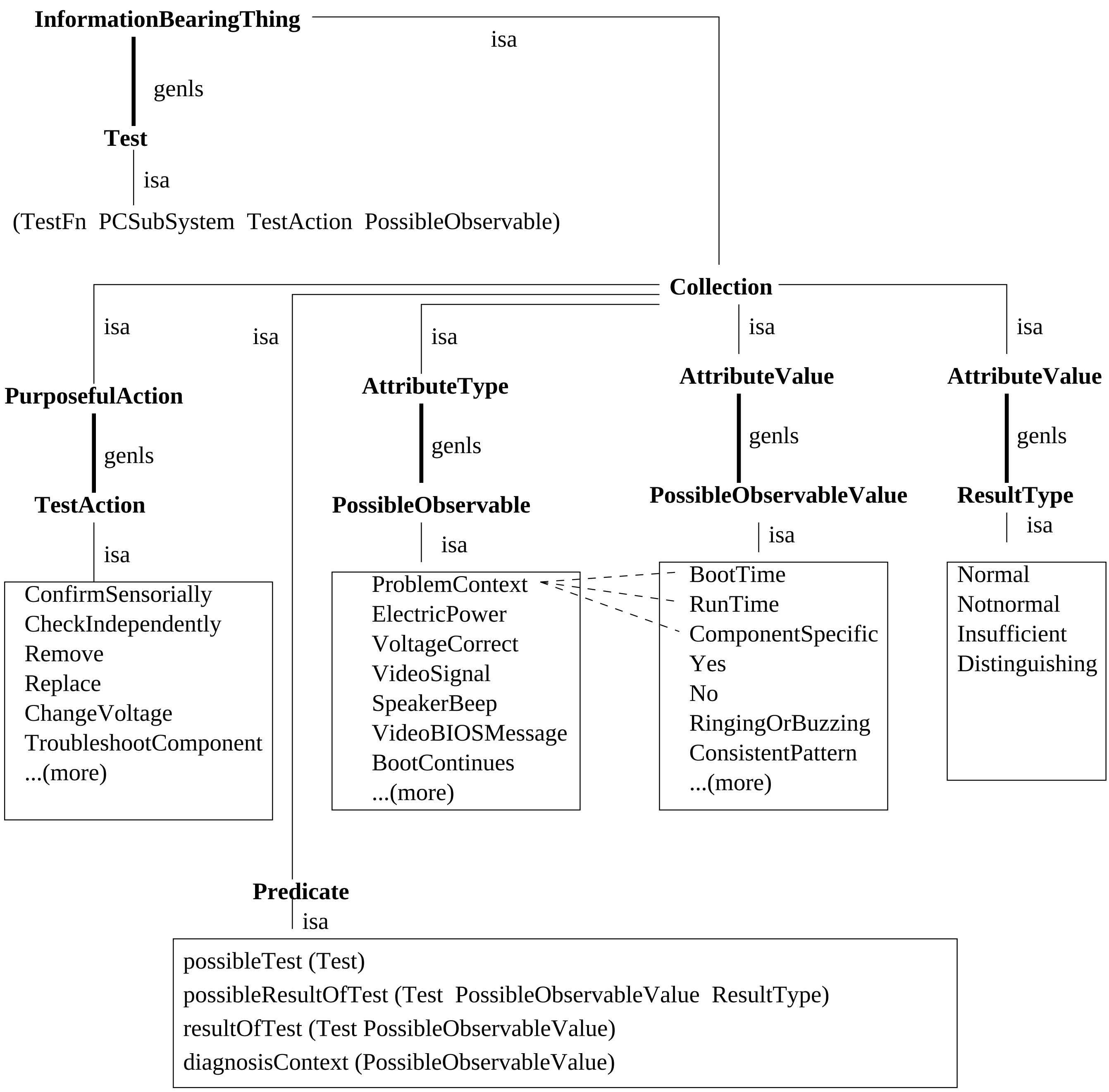

Figure 4.3: The Test Knowledge Hierarchy in CYC

(B:258), which takes as arguments the three constituent concepts, i.e. #$PCSubSystem, #$TestAction and #$PossibleObservable, and returns a #$Test instance. Schematically:

```
(TestFN PCSubSystem TestAction PossibleObservable) -> Test
```

Once again, the rules that introduce the possible tests, their possible results and their types, are context dependent:

```
(implies
 (and
  (diagnosisContext BootTime)
  (hypothesis HYPOTHESIS)
  (resultOfTest TEST_1 RESULT_1)
   ...
  (resultOfTest TEST_M RESULT_M))
 (and
  (possibleTest TEST)
  (possibleResultOfTest TEST POSSIBLE_OBSERVABLE_VALUE_1 RESULT_TYPE_1)
   ...
  (possibleResultOfTest TEST POSSIBLE_OBSERVABLE_VALUE_N RESULT_TYPE_N))).

where
  TEST:= (TestFn PC_SUBSYSTEM TESTACTION POSSIBLE_OBSERVABLE).
```

Example of Test introduction rule. The rule

```
(implies
 (and
  (diagnosisContext BootTime)
  (hypothesis MotherBoard)
  (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
  (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No))
 (and
  (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
```

```
  (possibleResultOfTest
    (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient)
  (possibleResultOfTest
    (TestFn MotherBoard ConfirmSensorially SpeakerBeep) ConsistentPattern
                                                         Insufficient))).
```

means that
IF

1. The problem solving context is Boot-time and
2. The Motherboard is hypothesised as being in fault and
3. Something is written on the screen (VideoSignal=Yes) and
4. There is no video BIOS message

THEN

- A possible test is to check if the speaker beeps and
- There are two possible results:
    1. No sound comes from the speaker, which is insufficient and necessitates more tests (about the speaker itself).
    2. A beep with a consistent pattern comes from the speaker, which is insufficient and necessitates more tests (about the beep code).

These rules are grouped in the KE-text by component/subsystem (B:763-1678).

4.4 The Implementation of the Inference and Task Layers

The guidelines for implementing the *control* knowledge, namely the Inference and Task layers, are:

1. These layers will be implemented in SubL code and

2. The implementation will allow for the problem-solving method, the Systematic Diagnosis PSM, to be executed *declaratively*, i.e. merely in terms of asking and (possibly) changing facts in the CYC KB.

Under these guidelines, the encoding of the Inference and Task layers in SubL code was done in the following way[4]:

1. Each inference type was encoded as a single SubL function-inference.

2. The only thing that these functions do is to ask the CYC KB questions (fi-ask), assert (fi-assert) and/or retract (fi-unassert) facts from the CYC KB (Inference layer) and - according to the new facts - call the appropriate function-inference (Task layer).

3. The Task Structure in figure 3.2 is encoded in a function, `systematic` (C:22), although this is not obvious from the SubL code for technical reasons that will be explained later.

The description of the implementation for the Inference and Task layer is based on figure 4.4. According to this figure, the overall system in CYC works as follows:

1. The function-inference `systematic` is called (Figure 4.5). When the diagnostic session begins, i.e. no #$PCSystem is hypothesised as faulty (number of hupotheses = 0), then the `sd-select1` (C:157) function is called. The latter, asserts to the KB the fact that the whole #$PCSystem is at fault (C:166). Each time that a new `(hypothesis ...)` assertion is entered into the KB, the appropriate rules concerning possible tests “fire”. This is because, in CYC, a rule can have a direction associated with it. If this direction is *forward*, then, whenever new assertions are asserted in the KB, the inference engine of CYC finds all rules with direction *forward* of which the antecedent part is satisfied by the contents of the KB and immmediately asserts all the assertions in the consequent part of it. This enables *forward reasoning* to be implemented, which is the case for the system developed in this research. In fact, forward reasoning is implemented by the Task structure.

[4]Numbers in parentheses refer to line numbers of the SubL listing in Appendix C.

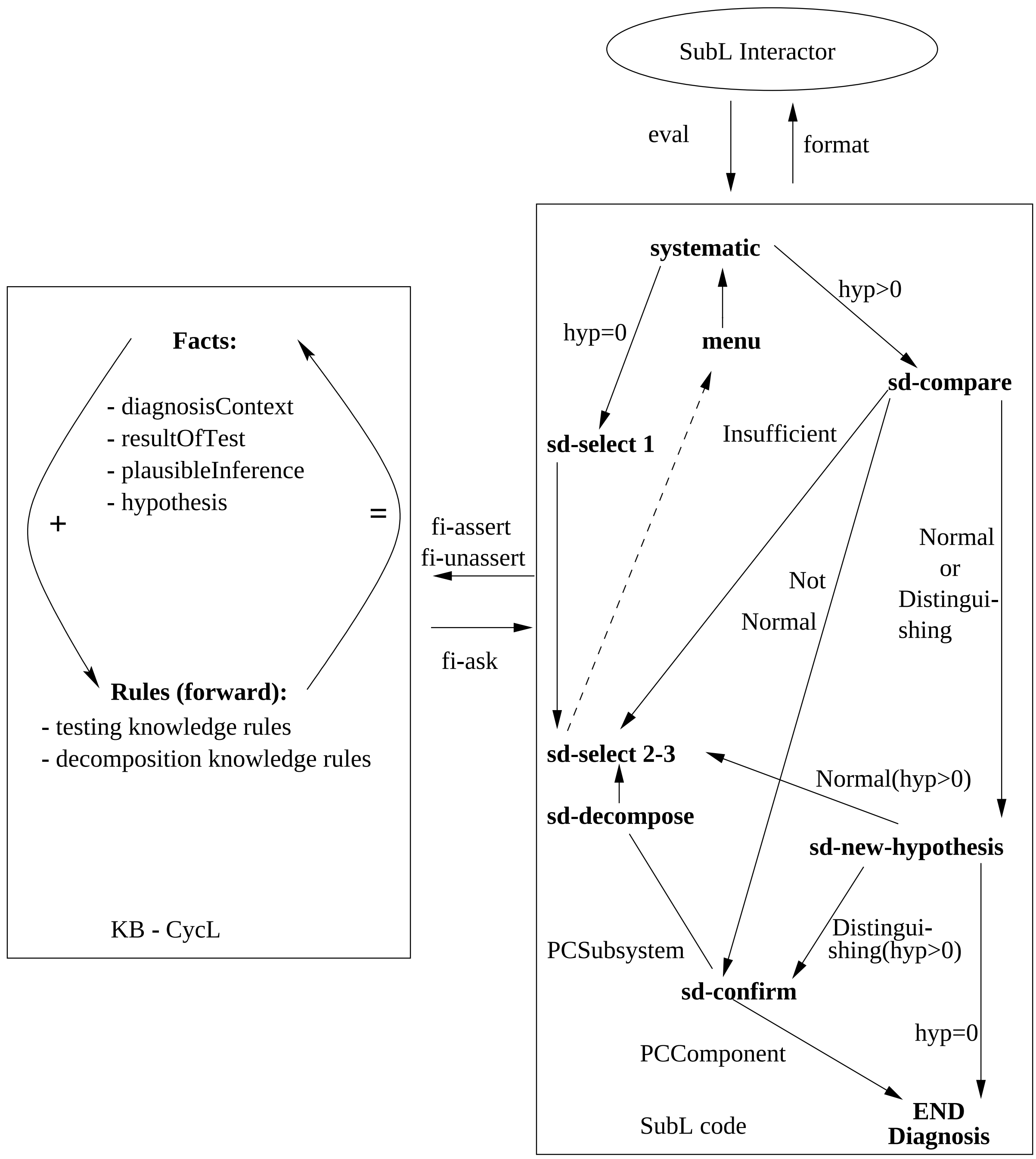

Figure 4.4: The Overall Model (detailed view)

The forward rules are involved only in answering questions to the KB and they can be replaced in the KE-text by the usual, backward rules of CYC, with minimal modifications to the SubL code[5].

2. The aforementioned rules assert in the KB facts about the possible tests that may be performed, their possible results and the types of these results. Therefore, the *select2-3* inference is actually implemented through these forward rules while the `sd-select2-3` (C:172) function is responsible only for asking the KB which is the next possible test and present it to the user in the SubL Interactor panel (Figure 4.6).

3. This is the place where a technical problem occured, which had a drawback and a benefit. The problem is that SubL code *cannot ask for direct input from the user* through the Interactor panel. This is a problem caused by the implementation of the CYC Web interface. Therefore, for the user to interact with the SubL code, the code must stop running and the user must enter the input needed by the code as a parameter to the `menu` (C:332) function(Figure 4.6). This is reflected in figure 4.4 by the dashed arrow to the `menu` function. The drawback now is that the code of the `systematic` (C:28) function does not reflect the Task Structure in figure 3.2, although the structure of the SubL code in figure 4.4 reflects the Task Structure when the `menu` function is removed and its calling functions are directly connected to the `systematic` function. The benefit is that, since the SubL code execution is interrupted and the control goes back to the CYC interface, the user can now go to other panels of the interface, like the Ask interface or the KB Browser interface and view the contents of the KB *as they have been modified* by the SubL code, with the following advantages:

   - The user can explicitly view which the current state in the diagnosis problem solving method is.

[5]A version of the SubL code that works with backward rules was developed but it is not included in this thesis, since the only modifications made were just adding some extra arguments to the calls of `fi-ask` function, that is, adding some parameters to the `fi-ask` function to perform backward chaining (see Appendix A).

- The user can use the justification mechanism of CYC to get a justification of the current state of the diagnosis.
- The user can even explicitly change the current state of the diagnosis by changing the contents of the KB and experiment with the system, although this will probably lead to system malfunction and would be recommended only to users experienced with the co-operation of the SubL code and the contents of the KB.

4. The `menu` function asserts into the KB the actual result of the last test performed and the result's type. It then calls function `systematic` to continue the diagnosis process. Having performed the *select2-3* inference, the next inference to be performed is *compare* and therefore the corresponding function, `sd-compare` (C:43) is called. This function decides which function-inference to perform next according to the result type:

   - If #$Normal or #$Distinguishing, then the subsystem which is the current hypothesis is not faulty and the next hypothesis, if one exists, must be considered (Figure 4.7). This is done by the `sd-new-hypothesis` (C:123) function which does not correspond to an inference but its a control function added as a result of the technical problem described above.
   - If #$NotNormal, then the subsystem which is the current hypothesis is faulty and it must be either decomposed, if it is a #$PCSubsystem (Figure 4.8), or located as the faulty component, if it is a #$PCComponent (Figure 4.9); that is what the `sd-confirm` (C:94) function does.
   - If #$Insufficient, then a new test must be performed and therefore, the `select2-3` function is called (Figure 4.10) .

It must be noted that the *decompose* inference is implemented through *forward* rules in the same way that the *select2-3* inference is implemented, while the `sd-decompose` (C:358) function performs only *managing* work with the KB by asking, asserting and retracting.

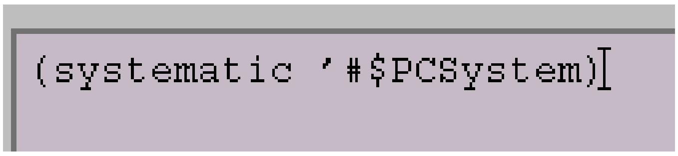

Figure 4.5: Starting the diagnosis session

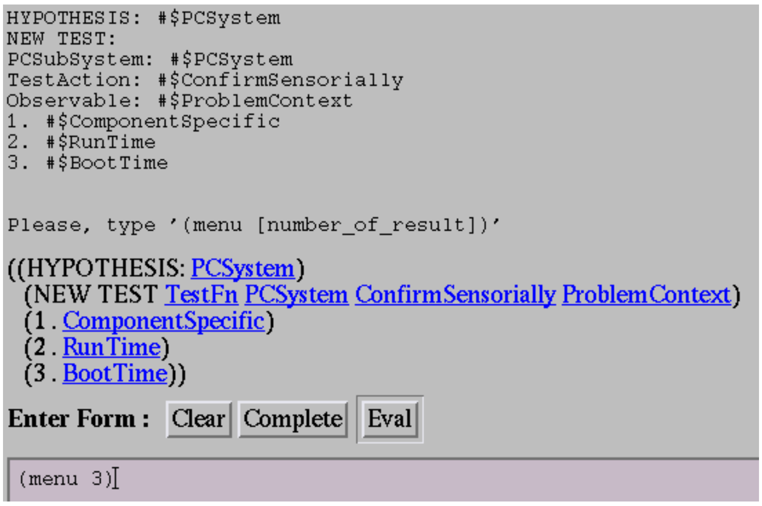

Figure 4.6: Presenting a Test to The User

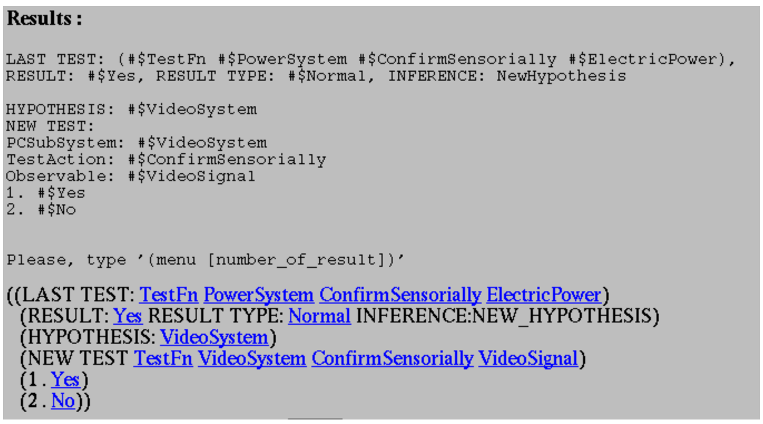

Figure 4.7: New Hypothesis After a “Normal” Result Type

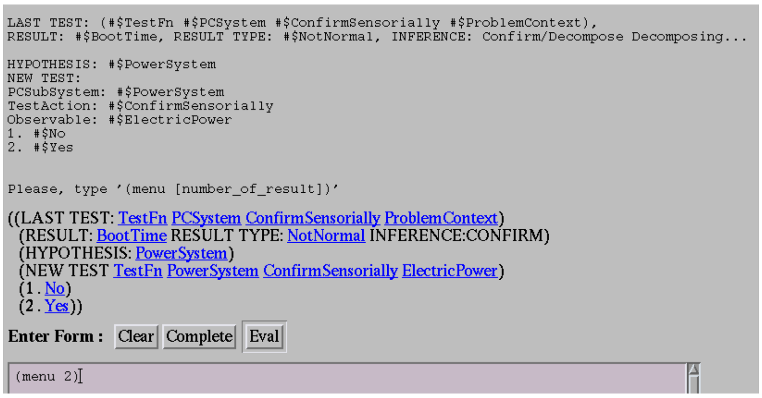

Figure 4.8: Decomposing a PCSystem After a "Not Normal" Result Type

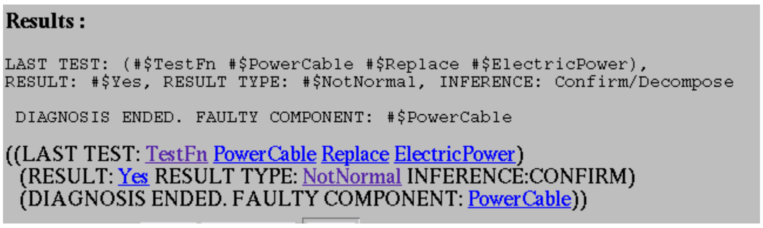

Figure 4.9: Confirming a PCComponent After a "NotNormal" Result Type

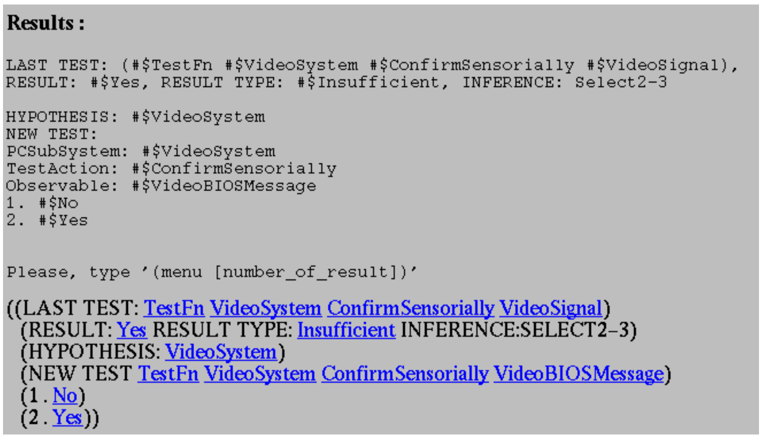

Figure 4.10: New Test After an "Insufficient" Result Type

In the SubL code listing in Appendix C, some more functions can be found. These are support functions and a brief description of them is given in table 4.1.

Function	**Description**
`get-test-result`	Presents the test to be performed by calling the appropriate functions. It provides to the user HTML links to every term of the test in the CYC KB (C:214) .
`present-test-parameters`	Presents the test parameters, that is, the #$PCSUbSystem, #$TestAction and #$PossibleObservable (C:246).
`present-test-results`	Presents the test's possible results (C:261).

`position-list`
 	
`get-ask-binding`
	Provide access to the bindings lists that function `fi-ask` returns. These lists are the ones returned when a question is made from the Ask interface (C:295, C:314).

`sd-reset`	Resets all the assertions concerning the systematic diagnosis problem solving method in CYC's KB. Must be called once before a diagnostic session begins (C:400).

Table 4.1: The SubL Support Functions

4.4.1 Maintaining the Declarative Scheme

A close examination of the SubL code in Appendix C for the function-inferences reveals that all they do is *managing* work:

1. they *ask* questions to the KB,

2. according to the results of these questions, they *retract* from the KB knowledge that is no more valid/necessary, *assert* knowledge that is valid/necessary and they *decide* which inference to perform next.

Therefore, *there is not any domain knowledge* hard-wired inside the code and the control knowledge is clearly seen. In turn, this means that a human user could perform the problem solving method equally well from the Ask interface, provided that he/she knew what question to ask to CYC each time and which assertions are valid/needed in each step of the method. This is the preservation of the declarative scheme in the implementation of the problem-solving method through SubL code.

4.5 Extending the system

The original goal of this research was to investigate the potential of using KADS as a methodology for developing problem solving methods (PSMs) in CYC. As far as it concerns KADS itself, the Generic Task Model (GTM) that forms the frame for developing the Expertise model is *independent* of the domain in which it is applied; it is only dependent on the task that must be performed in this domain. This in turn suggests that a significant part of the knowledge developed for a task in a specific domain should be easily re-usable for the same task in another domain. But the issue is, how easily? Moreover, with CYC given as the implementation environment, would this affect the ease of extending the GTM in another domain for the same task?

In order to answer these critical questions, the next step was to implement Systematic Diagnosis in another domain. The domain was Automobile fault diagnosis, restricted just to the Ignition system. The changes that occured are discussed below[6].

4.5.1 Changes of Microtheories

In the implementation of Systematic Diagnosis only for the PC domain, all the knowledge was grouped in a single microtheory, #$PCDiagnosisMt. However, this was done for reasons of simplicity. Despite this simplification, it was obvious, even from the analysis phase, that a great deal of the knowledge did not have to do with the PC domain but the problem solving method itself. For example, the general concepts of the testing knowledge (#$Test, #$TestAction, #$PossibleObservable, #$PossibleObservableValue) are concepts related with the Systematic Diagnosis PSM rather than with the specific domain. Therefore, by introducing another domain, this knowledge should be seperated from the PC domain knowledge, and it should also be available to the Automobile domain.

[6]Numbers in parentheses refer to the line numbers of the KE text listing in Appendix D.

In CYC terms, a different hierarchy of microtheories was needed. Two new microtheories were created, #$SystematicDiagnosisMt (D:9-18) and #$AutomobileDiagnosisMt (D:378-385), in addition to the already exisiting microtheory, #$PCDiagnosisMt (D:761-767). Of course, the latter two, are extensions of the first one, which in turn is an extension of the #$BaseKB microtheory; these relationships are given in Figure 4.11. It is obvious that there is a nice mapping between KADS and CYC: one microtheory corresponds to each domain and to each problem solving method.

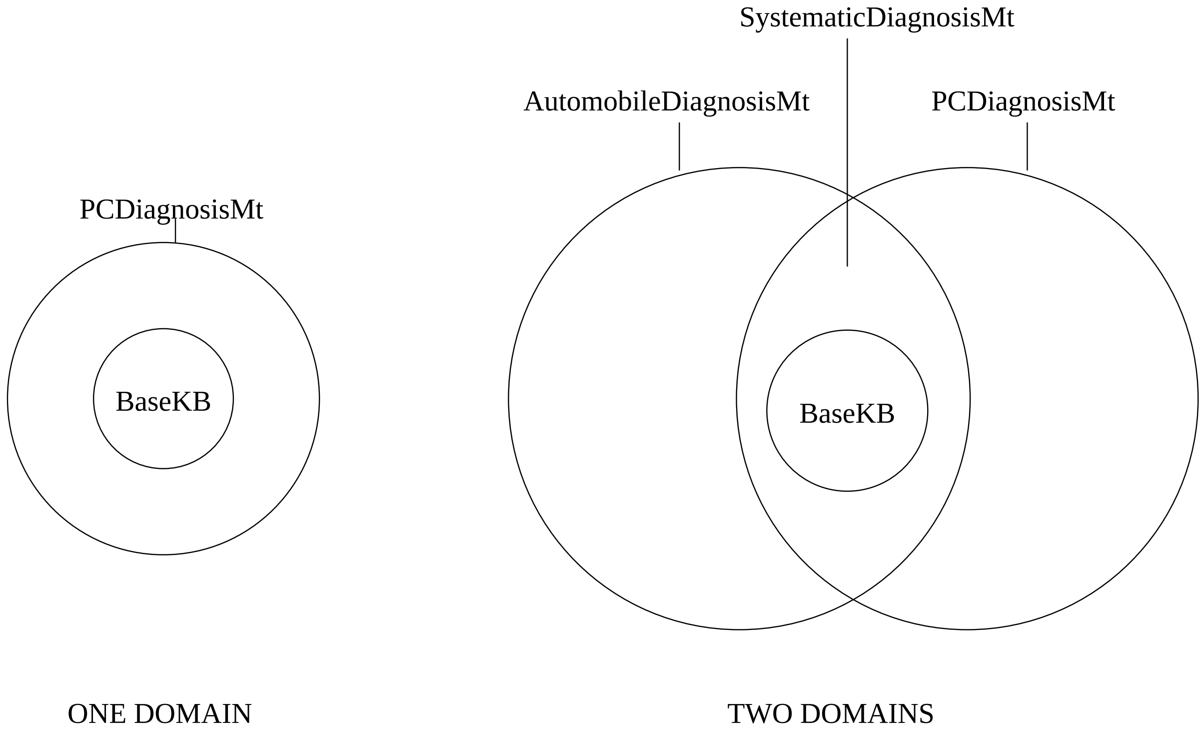

Figure 4.11: Changes in the Microtheories

4.5.2 Changes in the Domain Layer

Given the implementation of the Systematic Diagnosis PSM, and the three microtheories, the following changes have to be done in the domain layer (see Figure 4.12) :

1. Generalise concepts, relations and rules in the task-specific microtheory (#$SystematicDiagnosisMt). The generalisation is based on the common use of knowledge in both domains. Mainly this kind of knowledge is task and not domain dependent.

The generalisation may demand moving already existing domain structures from the domain-specific microtheories (#$PCDiagnosisMt and #$AutomobileDiagnosisMt), e.g. #$Test (D:106), #$hypothesis (D:65), to the task-specific one or creating new domain structures in the task-specific microtheory, e.g. #$SubSystem (D:26), #$Component (D:36).

2. Specialise concepts, relations and rules in the domain-specific microtheories. The specialisation is based either on connection of the domain knowledge with the task knowledge or on demands for special domain knowledge. In the first case, the new domain structures are specialisations of domain structures in the task microtheory, e.g. #$AutomobileSubSystem (D:395), #$PCComponent (D:787). In the second case, the new domain structures express domain-specific variations, e.g. #$physicalDecompositions (D:44, D:500-518) and #$functionalPartOf (D:925), which describe the system model in a structural and functional way correspondingly.

4.5.3 Changes in the Inference and Task Layers

As expected, the changes in the Inference and Task layers were minimal, practically ignorable. This fact is supported by the following:

1. The rules for testing and decomposition (D:536-738 for #$AutomobileDiagnosisMt and D:1208-2134 for #$PCDiagnosisMt), which implement the *select2-3* and *decompose* inference types, have exactly the same structure in both domains and

2. The SubL code, which implements the task structure, has changed only to differentiate the domain microtheory in which the KB operations (`fi-ask, fi-assert, fi-unassert`) are performed.

The fact that these two layers which constitute the GTM for Systematic Diagnosis were practically unchanged, implies that developing problem solving methods in CYC, with KADS as the developping methodology, provides modularity and ease of extension, knowledge re-use and analogy-driven knowledge acquisition between existing and new domains. These issues will be further discussed in the next chapter.

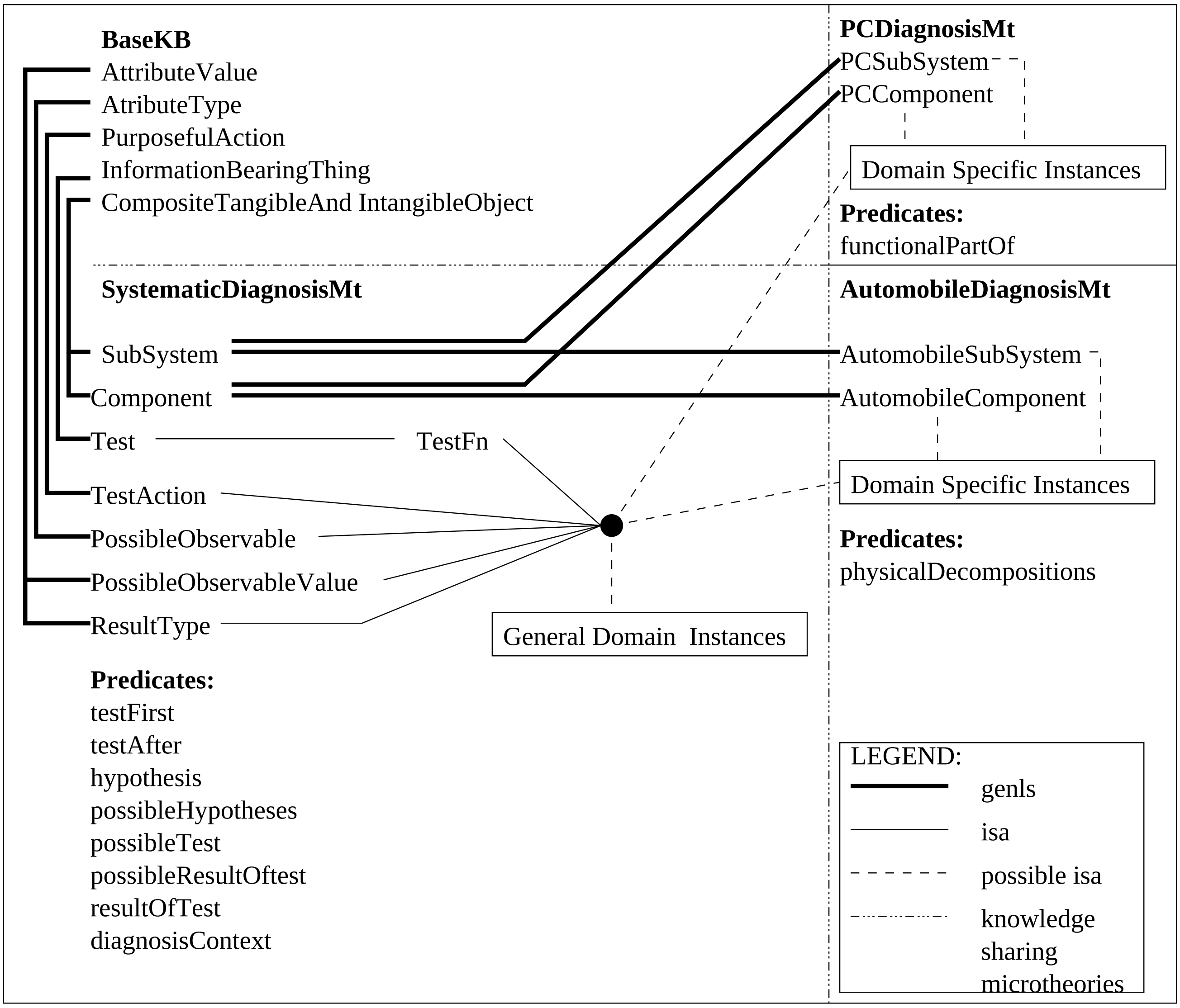

Figure 4.12: The Changes in the Domain Layers

4.6 Overview

In this chapter, the implementational decisions for the Systematic Diagnosis PSM in the domain of PC fault diagnosis were discussed. The main decisions were to implement the Domain layer in CYC's KB (using CycL) and the Inference and Task layers to be implemented in Lisp code (using SubL). The whole task is controlled by the Lisp code which operates (asks, asserts and retracts facts) on the KB through the Functional Interface. The main advantage of this implementation is that preserves to a considerable extent the *declarative scheme* of knowledge representation and inference in CYC. A small but conceptually essential extension of the system implemented in the domain of Automobile fault diagnosis was discussed, pointing out the main issues of reusing PSMs under the implementation developed.

Chapter 5

Issues and Results

Computers are useless. They can only give you answers.

- Pablo Picasso

An expert is a man who has stopped thinking - he knows!

- Frank Lloyd Wright

5.1 Introduction

The principal goal of this research was to implement in CYC a problem solving method (PSM) from the KADS methodology and study the issues that arise with this implementation. The principal issue to study was whether implementing a PSM in CYC would increase its reasoning power. Other issues were:

1. How well does KADS "fit" into CYC? KADS, as a methodology is implementation-independent. Its products, the various models from the analysis and design phases, can potentially be implemented in any implementation environment. CYC provides a declarative representation language, CycL, and a procedural/functional dialect of Lisp, SubL. The issue then is what part of CYC implements each product part of KADS and how.

2. Does KADS contribute to the development of CYC's Ontology and, if so, in what way?

But first, let us consider the first issue.

5.2 Problem Solving in CYC

The final objective of any intelligent agent is to solve problems. Intelligence itself is defined as the ability to solve problems. Expert Systems (ES) are the most developed problem solvers in the field of AI. However, first-generation ES suffered from what was called "the knowledge brittleness":

- They could not handle missing, incomplete or imprecise data.
- They could not give satisfactory explanations for what they were doing to solve a problem and how they were doing it.
- They could not handle situations which were not foreseen by their programmers and did not have explicit knowledge for them.

In general, the aforementioned problems could all be described as inability of the ES to fall back to "first principles", to use *common sense.* The motive for developing CYC was overcoming this "brittleness", this lack of common sense ([Lenat & Guha 90]). To build a system that could perform common sense reasoning, one should provide it *both* with a lot of common sense knowledge, facts and rules of thumb that an average person knows about the world, and common sense inferencing mechanisms like *analogical reasoning* that will perform on that knowledge. In their mid-term report book, [Lenat & Guha 90], the writers admit that the CYC project focused on the development of an *Ontology* of common sense knowledge, as

> ...successful analogizing depends on ... having a realistically large pool of (millions of) objects, substances, events, sets, ideas, relationships, etc, to which to analogize.

This focus to the common sense knowledge itself led to an imbalance as far as it concerns the problem solving abilities of CYC. The author's experience with CYC is that, although it provides a quite sophisticated environment for developing any kind of knowledge, it lacks the support for using that knowledge. This imbalance is explained in more detail below.

5.2.1 Inferencing in CYC

Currently, CYC starts its inferencing when a question, expressed in its representation language CycL, is asked from its Ask interface. To answer a question, CYC uses two kinds of inference mechanisms:

1. Backward chaining with resolution, as a general, "weak" inference mechanism.

2. Heuristic level, special-purpose inference mechanisms that decide which nodes to open at each stage of this backward chaining, that is which reasoning path to pursue next. These heuristic inference mechanisms are based on the syntactic form of each (sub)goal as well as on the occurrence of special predicates like #$isa, #$genls and others.

CYC also makes use of *microtheories* to restrict the search space. However, it is the author's opinion that these inference mechanisms are simply not enough to produce what CYC is intended to produce: *expertise*. "Weak" inference mechanisms are just to support expertise and not to produce it. A discussion about this issue can be found in [Luger & Stubblefield 98]. Moreover, the special-purpose heuristic inference modules only support the implementational level of inferencing and have nothing to do with the knowledge-level ([Newell 82]) which primarily concerns an ES. A discussion about this issue can be found in [Steels 90] and [Chandrasekaran 86]. Most important, the general, "weak" methods of inferencing, like backward reasonong with resolution, suffer heavily from the *combinatorial explosion* of the state space ([Luger & Stubblefield 98]), and this is more severe in the case of CYC's huge state space, its KB, containing 10^6 common sense axioms and still growing ([Lenat 95]). Therefore, it becomes obvious that what is missing from the inferencing power of CYC is what I would call an *Ontology of inference mechanisms*, that is, a large set of elementary inference mechanisms that perform elementary knowledge transformations and can be combined together to generate more complex inferencing for more complex problem-solving situations. In [Lenat & Guha 90] it is explicitly stated that such a set of inference mechanisms should be an indispensable part of any reasoning system:

> ...Breaking down the phenomenon [i.e. analogical reasoning] into its various subtypes and then handling each one.

5.3 Problem Solving Methods in CYC

Problem Solving Methods (PSMs), as they were described in this thesis, provide a source for this ontology of inference mechanisms. Although PSMs have been developed in a background broader than that of KADS, as general building blocks of expertise ([Chandrasekaran 86]), KADS methodology provides an elaborate organisation of these mechanisms, in the form of:

1. Primitive inference mechanisms, the *inference types*, with their elementary input and output, the *domain roles*

2. Structured entities of these primitive inferences, the *Generic Task Models.*

Whether this scheme is the best it is not of importance here. The important is that there are some elementary inferences and a way to combine them together to produce complex problem solving behaviour. It is obvious that this is the analogous of having a common sense knowledge base in the knowledge level, in another level (or meta-level), the inference level. Research in the development of ES has shifted to that level, either in the form of the *knowledge-use* level ([Chandrasekaran 86], [Steels 90]) or in the form of *proof plans* ([Bundy 88]). This shift cannot be ignored, especially in the case of CYC which was built as an ES development platform.

I would try a metaphor here to underline the importance of this level. I think that inference mechanisms are to intelligence what the natural laws are to physics and functions are to mathematics: they describe the fundamental interactions and transformations of the structural materials of each domain, namely of the knowledge, the matter/energy and the quantities correspondingly. This metaphor will be useful right below, where the implementation of the Systematic Diagnosis, the PSM selected for this thesis, is discussed.

5.3.1 Implementing the Systematic Diagnosis PSM

The above discussion gives an answer to the theoretical issue of *why* implement PSMs in CYC. In chapters 3 and 4 a thorough description of *how* this implementation was done is given.

The most important result of this implementation is that *the inferencing power of CYC improved*, as it can now perform systematic fault diagnosis not only in the domain of PCs, but in a range of domains as it was proved by the extension of the system in the domain of Automobiles (§4.5). The importance comes from the fact that CYC could not perform the same task by simply backtracking through its knowledge base; systematic fault diagnosis demands *dynamic* gathering of data through queries from the system to the user and consequent dynamic update of the knowledge base. These dynamic information collection and manipulation operations do not lend themselves to explicit declarative reasoning but naturally demand procedural and functional reasoning. Although CycL, the representation language of CYC, provides the means for calling LISP code (SubL), these means are not enough either to provide interaction with the user or to dynamically update the knowledge base of CYC. Hence, the implementation of the Task Structure, the overall plan of the PSM, in SubL (§4.2), and the use of the functional interface (FI) to ask and update the CYC KB *dynamically*, that is, assert or retract facts.

This latter fact, that simple backtracking and matching with the contents of the KB, in spite of its powerful generality, is not enough to implement any kind of problem solving task, proves in practice that a general reasoning system must provide both a variety of inference mechanisms and the means to combine and control them. The implementation of the Systematic Diagnosis problem solving method, as described in this thesis, establishes both a source for these mechanisms (KADS) and a way to implement them in CYC.

5.3.2 The Implemented System

Despite the lack of a variety of inference mechanisms, the CYC system provides an excellent platform for developing expert systems (ES). This is reflected in the expert system developed for this thesis that performs systematic fault diagnosis. The system has all parts of a typical ES:

- A menu-driven user interface, which additionally provides HTML links to any term in the KB, making the system self-explainable and documented by direct reference to its terms.

- Double inference engine: the built in backward chaining CYC inference engine for answering questions and the Task Structure encoded in SubL for performing the overall task of diagnosis.

- An explanation facility. Although not explicitly implemented, there is an explanation sub-system available through the `[Justify]` option that the CYC KB Browser provides for assertions in the CYC KB. Since the user can view the contents of the KB at any time during the diagnostic process, he/she can see the justification of any assertion in the KB in terms of which rules/facts prove the assertion. From this limited explanation facility, a more complex could be built by just accumulating in a list the justifications of every assertion relative to the diagnostic process. This feature can be implemented in the SubL code through the `fi-justify` function (see Appendix A).

- A KB editor, built in CYC.

- A general KB, CYC's #$BaseKB microtheory.

- Domain-specific knowledge, the #$PCDiagnosisMt and #$AutomobileDiagnosisMt microtheories.

The implemented system is itself a hybrid rule-based and model-based one (the model of the diagnosed system is implemented via rules), with all the advantages of such systems:

- Direct use of heuristic, unformalised diagnostic knowledge

- Modularity of rules

- Separation of domain and control knowledge which results to easiness of tracing, debugging and extending.

- Good performance in a *limited* domain.

The last characteristic of the system must be discussed more in detail as it touches on the motivating issue for building CYC itself. The motive was to overcome the "brittleness" of ES. This "brittleness" describes the inability of ES to handle novel or unexpected situations. Humans can handle these situations generally by falling back on "first principles" or general (common sense) knowledge. CYC was build to provide this common sense knowledge as a common substrate for the development of ES. Therefore, one would expect that the system that was developed in this thesis and any other ES that would have been developed in CYC would not suffer from this "brittleness". This is not the case by any means. The system of this thesis and any other ES developed in CYC would suffer from "brittleness" *unless it was explicitly designed* to take advantage of CYC's common sense ontology. The reason for this is simple. An ES has as its purpose to produce the same behaviour as a human expert. But, most of the time, a human expert uses *heuristic knowledge*, that is knowledge which is a "distilled" part of both the general and domain-specific knowledge which has proved to be the most important and useful for the expert to perform his task. Of course, the human expert is completely aware *how* he formed this heuristic knowledge and, in case it is not directly applicable for reasons of novelty, can review it and reason about it. But to reason about his heuristic knowledge, the human expert must use the original knowledge from which he formed the heuristic one. These two levels of knowledge are known in the theory of ES as *surface and deep* knowledge correspondingly ([Steels 90]). This is where CYC comes. It provides the "deep" knowledge that is needed for an ES to reason about its heuristic one. But the ES must be designed to be able to perform that kind of reasoning, that is reason about its heuristic knowledge using general knowledge. An example from the actual system will make things clear.

In order to perform diagnosis in PCs, the system has heuristic rules about the *order* of diagnosis. This is because the order is important for the diagnosis (see §3.4.1). Of course, the system does not know why the order in its rules is the one encoded. If, for some reason, this ordering could not be applied, then the system is unable to reason. But, if it should be able to reason about the rules themselves, that is check them for consistency, alter them or even infer them, then more knowledge would be needed. Specifically, in the case of the ordering rules, the system would probably need some or all of the following:

1. A structural model of the system, the way the components are connected together to form the various (sub)systems. This model by itself would need a complete ontology, like the one found in [Borst *et al.* 97].

2. A theory of "ordering" in testing, e.g. easiness, that would require certain criteria (rules) of which systems are easy to test:

- Systems that can be tested sensorially (visually, acoustically) rather than by instruments.
- Systems that are easy to replace.
- Systems that are easy to test in isolation.
- Systems with simple structure (Components vs. Sub-systems)
- Systems that are easy to access (e.g. terminal systems)

Note that the knowledge outlined above could need more knowledge like knowledge about serial and parallel connections, control devices, input and output of devices, etc.

It is obvious that for just a simple group of rules, the rules of ordering, the knowledge of the system would have to grow dramatically. And this can happen for every part of the system's heuristic knowledge: what test to perform, how to perform it, what system variable to test, what makes a system variable to be such and so on. The obvious conclusion is that to overcome "brittleness" two things are needed:

1. A large (huge in the case of CYC) general (and less general) knowledge base and

2. An elaborate connection of the heuristic knowledge of the ES with the appropriate part of the general knowledge.

Consequently, even with CYC's upper ontology in hand, it requires a tremendous effort to build the part of general knowledge that an ES needs to reason about its heuristic knowledge and then to put things together, that is design when, why and how the system will fall back to general knowledge[1].

With the discussion of the "brittleness" problem concludes the discussion about the main issue of this thesis and the corresponding results. However, in the introduction of this chapter two secondary issues were mentioned, concerning KADS and CYC. These issues are discussed in the next section.

[1]This situation is well described in Lenat's report for using CYC in the High Performance Knowledge Base project, which can be found in `http://www.cyc.com/hpkb/proposal-summary-hpkb.html`

5.4 KADS and CYC

Problem solving methods (PSMs) were developed in the context of ES development methodologies, such as KADS. Therefore, PSMs can come from somewhere else than KADS. However, KADS, as a complete ES building methodology, provides the following as far as it concerns PSMs:

- A hierarchy of PSMs according to the task they are appropriate to (see Figure 2.5).
- A library of more elemenatry inference types that can be combined to build new PSMs (see Figure 2.1).
- Domain roles for the various inference types that guide the knowledge acquisition process (see Table 3.1).
- A useful distinction of the four layers of a PSM (domain, inference, task and strategy) and their interrelationships in the Expertise model which they form (see Figure 2.4).

It is obvious that KADS provides much more than a simple source of PSMs and therefore it is worth investigating how much KADS fits into CYC.

The first issue is that of the modelling process of KADS. As described in chapter 2, this is the Expertise model, consisting of four layers, the Domain, Inference, Task and Strategy ones. In chapter 4, a mapping between these layers and some parts of CYC was given, that is:

1. The Domain layer is implemented as a microtheory in CYC KB.
2. The Inference layer is implemented partly in the KB in the form of rules, as far as it concerns the transformation of the domain structures, and partly in the SubL code as functions-inferences, as far as it concerns the management of the transformed domain structures, that is retrieving, asserting and retracting them.
3. The Inference layer is implemented in the SubL code which has the overall inferencing control of the PSM.

4. Knowledge that is specific to the PSM is encoded in the KB in a seperate micro-theory.

The advantages of this mapping is that it distinguishes between the various kinds of knowledge needed to perform the task of the PSM and that it preserves to a great extent the declarative scheme of CYC (§4.4.1).

The second issue is how much KADS contributes to the development of CYC's ontology. From the description of the Analysis phase (chapter 3), the system extension (§4.5) and the issue of "brittleness" (§5.3.2), it becomes obvious that KADS provides an excellent methodology for developing CYC's ontology in a bottom-up manner, that is from the heuristic, surface, task-specific knowledge to the common sense, deep, task-independent knowledge, through the development of intermediate ontologies that will fill the gap between these two kinds of knowledge, as shown in Figure 5.1.

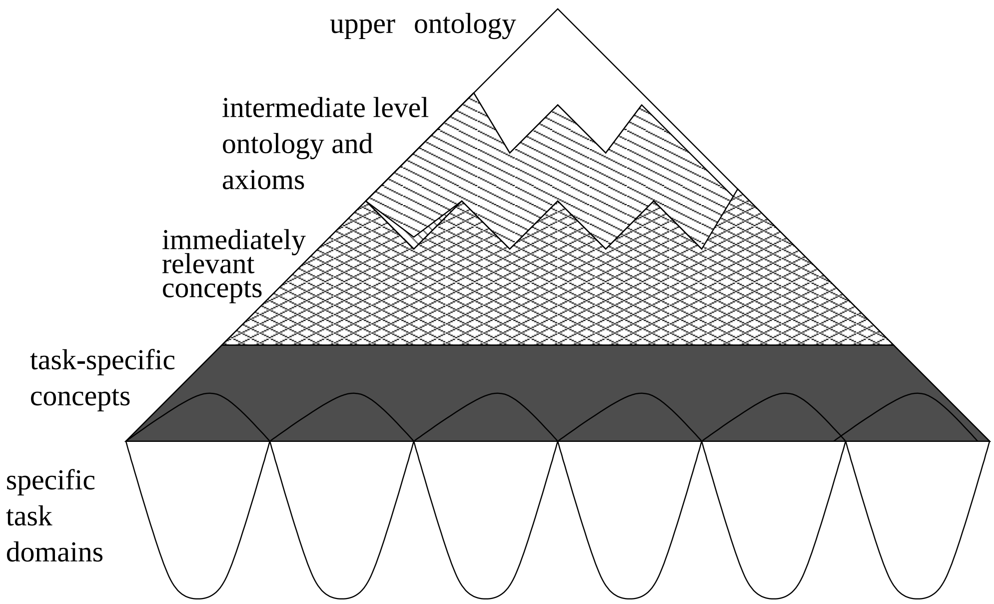

Figure 5.1: Filling the Knowledge Gap

In [Lenat & Guha 90] it is explicitly mentioned that CYC's ontology should be neither an encyclopedia of linearly arranged terms nor a thesaurus of disconnected knowledge. The main point of KADS is that it views ES building not as filling a pool of knowledge but as modelling the expertise behaviour according to specific tasks. This approach is also justified by the way humans acquire their expertise: by tackling a large diversity of tasks which continuously grow in complexity. Most of our knowledge is task-oriented. It is acquired and connected with the rest of the already existing knowledge that we have acquired, according to how, when and for what purposes it is useful. This is the approach on which KADS is based: KADS supports this approach both by its general modelling directions and the *domain roles* that introduces. On the other hand, KADS is a complete methodology that supports the whole cycle of an ES development, from the knowledge acquisition ([Kingston 94]), to analysis and design ([Tansley & Hayball 93]); it is even possible to use KADS for small ES as described in [Kingston 92]. It is therefore obvious how much KADS can contribute to the development of CYC's ontology in a bottom-up manner. Of course, the top-down direction of developing a global ontology is also useful and necessary, with its own advantages and disadvantages ([Guarino 98], [Smith 98], [Varzi 98]).

5.5 Further Work

It was explained before in this thesis that the implemented system was kept to the minimum necessary to investigate the main issue, which was the implementation of PSMs in CYC. However, a number of possible extensions can be considered:

1. Extending the domain knowledge: in section 3.2, it is mentioned that although the "Expert" used to make the knowledge acquisition included three troubleshooting contexts for PCs, only the Boot-time context was analysed. It would be a good test for the implementation to be expanded in the other two domains: the Run-time and Component-specific contexts. In the case of Automobiles this is even more necessary, since only a small part of the knowledge about the ignition system was analysed.

2. Developing some "deep" knowledge: in section 5.3.2, in the discussion about the "brittleness" of the system, it was mentioned that a kind of "ordering theory" would be needed if the system should be able to reason about the ordering of the tested systems. Some rules were given but more elaborate work should be carried out.

3. Implementing another PSM either for the same task or for a different one: it was mentioned that Heuristic Classification could be used for troubleshooting, while a configuration task could probably also make use of some of the knowledge encoded for troubleshooting. Implementing PSMs either for the same task or for a different one could promote the investigation of issues of knowledge re-use from another PSM and of how well the decisions for the implementation of Systematic Diagnosis work for another PSM.

4. The three previous extensions would provide a good paradigm for investigating how easily KADS contributes to the development of intermediate ontologies in CYC.

5. A more elaborate justification mechanism could be developed, by accessing CYC's built-in justification mechanism, keeping the necessary justifications for every inference step during the diagnosis process.

6. Although improving its reasoning power, the various parts of the PSM like the *inference types*, *inference structure*, *task structure* and *generic task model* are not known to CYC since they were implemented in SubL code. It could be a very interesting extension to investigate to what degree all these could be encoded *declaratively* using CycL, and consequently be used by CYC in its usual reasoning schema. A possible application could be to develop an ES that would guide a knowledge engineer to use KADS for developing knowledge-based systems, as described in [Kingston 95].

5.6 Conclusions

From the discussion in this chapter, the following conclusion come out of this thesis:

1. The inference mechanism of CYC, backward chaining with resolution, is not strong enough to perform more complex tasks that require dynamic information collection and KB updating, as is the case in the systematic diagnosis of faults in PCs. Moreover, this "weak" inference mechanism suffers from the *combinatorial explosion* problem.

2. The implementation of problem solving methods (PSMs) reinforces the inferencing power of CYC and enriches it with new inference mechanisms.

3. KADS methodology provides a rich source for PSMs and a structured way for using them in every aspect of problem solving, from knowledge acquisition to analysis and design of the problem solving model.

4. Moreover, KADS provides a systematic, task-oriented way for developing CYC's Ontology in a bottom-up way: from the task-specific concepts to the task-independent ones which form CYC's Upper Ontology.

All these conclusions should motivate further research in combining these two very promising state-of-the-art components of Artificial Intelligence.

Bibliography

[Blair *et al.* 92] P. Blair, R.V. Guha, and W. Pratt. Microtheories: An Ontological Engineer's Guide. Cyc Technical Report CYC-050-92, Microelectronics and Computer Technology Corporation (MCC), March 1992.

[Borst *et al.* 97] P. Borst, H. Akkerman, and J. Top. Engineering Ontologies. *International Journal of Human Computer Studies*, 46(213), 1997.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *Ninth Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.

[Chandrasekaran 86] B. Chandrasekaran. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. *IEEE Expert*, 1(3), 1986.

[Clancey & Letsinger 82] W.J. Clancey and R. Letsinger. Neomycin: Reconfiguring A Rule-Based Expert System for Application to Teaching. Technical Report STAN-CS-82-908, Department of Computer Science, Stanford University, May 1982.

[Clancey 85] W.J. Clancey. Heuristic Classification. *Artificial Intelligence*, (27), 1985.

[Guarino 98] N. Guarino. Formal Ontology and Information Systems. In N. Guarino, editor, *Proceedings of the First International Conference on Formal Ontology in Information Systems*, pages 3–15. IOS Press, 1998.

[Hamscher 88] W.C. Hamscher. Model-Based Troubleshooting of Digital Systems. Technical Report AI-TR 1074, Artificial Intelligence Laboratory, MIT, August 1988.

[Hesketh & Barett 90] P.H. Hesketh and T. Barett. An Introduction to the KADS Methodology. ESPRIT P1098, Deliverable M1, STC Technology Ltd., March 1990.

[Kingston 92] J. Kingston. PragmaticKADS: A methodological approach to a small knowledge based systems project. Technical Report AIAI-TR-110, Artificial Intelligence Applications Institute, University of Edinburgh, November 1992.

[Kingston 94] J. Kingston. Linking Knowledge Acquisition with CommonKADS Knowledge Representation. Technical Report AIAI-TR-156, Artificial Intelligence Applications Institute, University of Edinburgh, July 1994.

[Kingston 95] J. Kingston. Applying KADS to KADS: knowledge based guidance for knowledge engineering. Technical Report AIAI-TR-158, Artificial Intelligence Applications Institute, University of Edinburgh, January 1995.

[Lenat & Guha 90] D.B. Lenat and R.V. Guha. *Building large knowledge-based systems. Representation and in-*

ference in the Cyc project. Addison-Wesley, Reading, Massachusetts, 1990.

[Lenat 95] D.B. Lenat. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11), November 1995.

[Luger & Stubblefield 98] G.F. Luger and W.W. Stubblefield. *Artificial Intelligence. Structures and Strategies for Complex Problem Solving. 3rd ed.* Addison-Wesley, 1998.

[Newell 82] A. Newell. The Knowledge Level. *AI Journal*, 19(2), 1982.

[Rademakers & Vanwelkenhuysen 93] P. Rademakers and J. Vanwelkenhuysen. Generic Models and their Support in Modelling Problem Solving Behaviour. In Simons R. David, J.M. and J.P. Krivine, editors, *Second Generation Expert Systems*. Springer-Verlag, 1993.

[Schreiber *et al.* 93] A. Th. Schreiber, B. J. Wielinga, and J.A. Breuker. *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, London, 1993.

[Smith 98] B. Smith. Basic Concepts of Formal Ontology. In N. Guarino, editor, *Proceedings of the First International Conference on Formal Ontology in Information Systems.* IOS Press, 1998.

[Steels 90] L. Steels. Components of Expertise. *Artificial Intelligence Magazine*, 1990.

[Tansley & Hayball 93] D.S.W. Tansley and C.C. Hayball. *Knowledge-Based Systems Analysis and Design. A KADS Developer's Handbook.* Prentice Hall, London, 1993.

[Vanwelkenhuysen 92] J. Vanwelkenhuysen. Scaling-up Model-Based Troubleshooting by Exploiting Design Functionalities. In *Proceedings of the Fifth International Conference on Industrial and Engineering Applications and Expert Systems*. Springer-Verlag, 1992.

[Varzi 98] A.C. Varzi. Basic Problems of Mereotopology. In N. Guarino, editor, *Proceedings of the First International Conference on Formal Ontology in Information Systems*, pages 29–38. IOS Press, 1998.

[Wielinga *et al.* 92] B.J. Wielinga, A.Th. Schreiber, and J.A. Breuker. KADS: a modelling approach to knowledge engineering. *Knowledge Acquisition*, (4), 1992.

Appendix A

The CYC FI Function Reference

Core Functions

```
fi-assert : formula mt &optional (el-tv :default) direction -> [boolean]

     function:
         Add a local argument for formula to the KB within microtheory mt.
         el-tv is either :default or :monotonic.
         direction is either :forward, :backward, or NIL.
         If direction is NIL,
              ground formulas are entered as :forward,
              rules are entered as :backward.
         If formula is already present in mt with a different "hl truth value" (:true-monotonic, :true-default,
         :false-monotonic, or :false-default), change it to the new "hl truth value" determined by formula
         and el-tv.
         If formula is already present in mt with a different direction, its direction is changed.
         If direction is :forward, and formula was not already present in mt with a forward direction, then
         forward inference is performed on the formula.
     return:
         NIL if operation had an error.
         T if operation succeeded.
     errors:
         :arg-error
              one of the arguments was invalid
         :not-well-formed
              formula was not well-formed
     warnings:
         :redundant-local-assertion
              the assertion is already in the KB locally
         :change-local-tv
              the assertion is already in the KB locally with a different truth value

fi-unassert : formula mt -> [boolean]

     function:
         Remove any local argument for formula within mt.
     return
         NIL if operation had an error.
         T if operation succeeded.
     errors:
         :arg-error
              one of the arguments was invalid
         :not-well-formed
              formula was not well-formed
     warnings:
```

```
        :assertion-not-present
             the formula is not in the KB at all
        :assertion-not-local
             the formula is in the KB, but has no local support
```

fi-justify : formula mt &optional full? -> [argument]

```
    function:
        Provide an argument justifying belief in formula within mt.
        If full? is NIL, only provide one level of argument, which may include non-ground facts.
        If full? is non-NIL, follow argument down as far as necessary to reach ground facts.
    return
        NIL if operation had an error.
        NIL if it could not be justified.
        Argument if it could be justified.
             [argument] ::= list of [support]
             [support] ::= ([module] [fomula] [mt])
             [module] is one of :axiom :isa :genls :equality :eval :reflexive :symmetric :transitive :external
             [fomula] is the formula of the support
             [mt] is the microtheory of the support
    errors:
        :arg-error
             one of the arguments was invalid
        :not-well-formed
             formula was not well-formed
    warnings:
        :assertion-not-present
             the formula is not in the KB at all
```

fi-ask : formula mt &optional backchain? number time depth -> [ask-result]

```
    function:
        Ask for bindings for free variables which will satisfy formula within mt.
        If backchain? is NIL, no inference is performed.
        If backchain? is T or an integer, inference is performed.
        If backchain? is an integer, then at most that many backchaining steps using rules are used.
        If number is an integer, then at most number bindings are returned.
        If time is an integer, then at most time seconds are spent on the operation.
        If depth is an integer, then the inference paths are limited to depth total steps.
    return
        NIL if operation had an error.
        An ask-result if operation completed.
        [ask-result]
             ::= list of [bindings]
             ::= (((T . T))) if there were no free variables
        [bindings] ::= list of [binding]
        [binding] ::= ([variable] . [value])
        [variable] is a free variable from the formula
        [value] is the binding satisfying the formula
    errors:
        :arg-error
             one of the arguments was invalid
        :not-well-formed
             formula was not well-formed
```

fi-continue-last-ask : &optional backchain? number time depth reconsider-deep ->
[ask-result]

```
    function:
        Continue the last ask that was performed with more resources.
        If backchain? is NIL, no inference is performed.
        If backchain? is t or an integer, inference is performed.
        If backchain? is an integer, then at most that many rules are used.
        If number is an integer, then at most number bindings are returned.
        If time is an integer, then at most time seconds are spent on the operation.
        If depth is an integer, then the inference search cuts off at depth.
```

```
        If reconsider-deep is non-nil, then previous inference paths which were cut off for going past depth
        are reconsidered.
    return
        NIL if operation had an error.
        An ask-result if operation completed.
        [ask-result]
             ::= list of [bindings]
             ::= (((T . T))) if there were no free variables
        [bindings] ::= list of [binding]
        [binding] ::= ([variable] . [value])
        [variable] is a free variable from formula.
        [value] is the binding satisfying formula.
    errors:
        :arg-error
             one of the arguments was invalid
        :not-well-formed
             formula was not well-formed

fi-ask-status : -> [ask-status]

    function:
        Explain why the last ask completed.
    return
        an ask-status which is
             :EXHAUST if the search space was exhausted.
             :DEPTH if the search space was exhausted, but some nodes were too deep.
             :NUMBER if the requested number was reached.
             :TIME if the time alloted expired.
```

Appendix B

The CYC KE-Text for PC Domain

```
;;; PROJECT: 628-Implementing   Problem Solving Methods  (PSMs) in Cyc
;;; FILENAME:  diagnosisKE.txt
;;; AUTHOR: Dimitrios  Sklavakis
;;; PURPOSE: Contains  Cyc's   Knowledge Entering (KE)   text defining the
;;; knowledge  (Domain and  Expertise)  for   the  implementation  of   the
;;; 'Systematic   diagnosis'  PSM  from KADS methodology,    for PC faults
;;; diagnosis.

;;; LAST UPDATED: 04/08/1998.


;**************** DOMAIN KNOWLEDGE ****************

;**************** THE PC DIAGNOSIS MICROTHEORY ****************

;;; The  whole knowledge for performing   PC fault diagnosis  will be
;;; entered in the PCDiagnosisMt microtheory, which is a more specific
;;; microtheory of   the BaseKB microtheory:

;;;  (#$genlMt #$PCDiagnosisMt #$BaseKB).

constant: PCDiagnosisMt.
isa: Microtheory.
genls: BaseKB.
comment: "#$PCDiagnosisMt is the #$Microtheory that contains all the assertions
about performing Personal Computer(PC) fault diagnosis.".

; **************** THE SYSTEM MODEL ****************

Default Mt: PCDiagnosisMt. ;Change default microtheory to #$PCDiagnosisMt.

constant: PCSubSystem.
isa: Collection.
genls: CompositeTangibleAndIntangibleObject.
comment: "The collection of all PC sub-systems, like the
#$VideoSystem, #$PowerSystem, #$KeyboardSystem. Each instance of
#$PCSubSystem may include several #$PCComponents and/or other
#$PCSubSystems. Different #$PCSubSystems may include the same
#$PCComponents. In the context of #$PCDiagnosisMt any #$PCSubSystem is
an intermediate level of analysis for the #$PersonalComputer; the
diagnosis continues until a faulty #$PCComponent is located".

constant: PCComponent.
isa: Collection.
genls: PCSubSystem.
comment: "The collection of all PC components such as the
#$PowerSupply, #$VideoCard, #$FloppyDiskDrive. In the context of
```

```
#$PCDiagnosisMt any #$PCComponent is the lowest level of analysis for
the #$PersonalComputer; the diagnosis terminates when a faulty
#$PCComponent is located.".

constant: PCSystem.
isa: Individual PCSubSystem.
comment: "The #$PCSystem is used to refer to the #$PersonalComputer as
a #$PCSubSystem. It includes the following #$PCComponents:
#$PowerSystem, #$VideoSystem, e.t.c.".

constant: PowerSystem.
isa: Individual PCSubSystem.
comment: "The power #$PCSubSystem. Includes the following
#$PCComponents: #$PowerSocket, #$PowerCable, #$PowerProtectionDevice
(optionally) and  #$PowerSupply.".

constant: PowerSocket.
isa: Individual PCComponent.
comment: "#$PowerSocket is the socket that provides electric power to
the PC. It is a component of the #$PowerSystem.".

constant: PowerCable.
isa: Individual PCComponent.
comment: "#$PowerCable is the cable that connects the #$PowerSocket
with the #$PowerSupply. It is a component of the #$PowerSystem.".

constant: PowerProtectionDevice.
isa: Individual PCComponent.
comment: "#$PowerProtectionDevice is any device (supproccessor, UPS)
connected between the #$PowerSocket and the #$PowerSupply to protect
the #$PersonalComputer from power failures. It is an optional
component of the #$PowerSystem.".

constant: PowerSupply.
isa: Individual PCComponent.
comment: "#$PowerSupply is the component located inside the
#$PersonalComputer case that supplies the #$MotherBoard with electriv
power. It is a component of the #$PowerSystem.".

constant: VideoSystem.
isa: Individual PCSubSystem.
comment: "The video #$PCSubSystem. Includes the following
#$PCComponents: #$MotherBoard, #$VideoCard, #$Monitor (and the
#$Speaker).".

constant: VideoCard.
isa: Individual PCComponent.
comment: "The #$VideoCard trasforms the video information to video
signal and sends it to the #$Monitor. It is a component of the
#$VideoSystem.".

constant: Monitor.
isa: Individual PCComponent.
comment: "The #$Monitor trasforms the video signal sent by the
#$VideoCard into visual image. It is a component of the
#$VideoSystem.".

constant: MotherBoard.
isa: Individual PCComponent.
comment: "The #$MotherBoard is the main #$PCComponent. Most of the
rest #$PCComponents are connected onto the #$MotherBoard and
controlled by it. It is actually a sub-system by itself as it includes
other components but in the context of PC diagnosis it will be
regarded as a #$PCComponent to avoid increasing the complexity of the
#$PersonalComputer analysis.".
```

```
constant: Speaker.
isa: Individual PCComponent.
comment: "The PC speaker. It refers to the
internal speaker that is connected on the motherboard and not the
external ones that are part of a multi-media system and require a
sound-card on which they are connected.".

constant: BIOSStartupSystem.
isa: Individual PCSubSystem.
comment: "The BIOS startup #$PCSubSystem. Includes the following
#$PCComponents: #$MotherBoard, #$VideoCard.".

constant: BIOSsettings.
isa: Individual PCComponent.
comment: "The Basic Input Output System (BIOS) settings record the
system parameters for all its operations. Wrong BIOS settings can be
responsible for a PC malfunction, e.g., the #$FloppyDiskDrive being
disabled and therefore being non-existant to the system.".

constant: MemorySystem.
isa: Individual PCSubSystem.
comment: "The memory #$PCSubSystem. Includes the following
#$PCComponents: #$RAM, #$MotherBoard.".

constant: RAM.
isa: Individual PCComponent.
comment: "The Random Access Memmory #$PCComponent. Usually, it is a
set of Single In-Line Memory Modules (SIMMs), plugged in special
positions on the #$MotherBoard.".

constant: FloppySystem.
isa: Individual PCSubSystem.
comment: "The #$FloppySystem consists of the #$FloppyDiskDrive and the
#$BIOSsettings for the enabling/disabling of the #$FloppyDiskDrive.".

constant: FloppyDiskDrive.
isa: Individual PCComponent PossibleObservableValue.
comment: "The PC storage device which drives a removable floppy disk to
store/retrieve information.".

constant: HardDiskDrive.
isa: Individual PCComponent PossibleObservableValue.
comment: "The PC main mass storage device. Usually it is fixed and
non-removable.".

constant: CDROMdrive.
isa: Individual PCComponent.
comment: "".

constant: PlugAndPlaySystem.
isa: Individual PCSubSystem.
comment: "This system comprises all peripherals that are connected to
the PC via expansion cards and they are (usually) automatically
recognised by MS Windows without any extra software drivers or
configuration procedures. However, sometimes there may be some
problems with their recognition. This #$PCSubSystem may has as
functional parts a wide variety of peripherals. In the current
implementation of Systematic diagnosis, it is regarded as consisting
of peripherals and their #$ExpansionCards. There isn't any further
decomposition into the specific peripherals as these are vary in each
configuration.".

constant: ExpansionCard.
isa: Individual PCComponent.
comment: "This PC component is used in conjunction with various
peripherals. In the current implementation, it is regarded as a
```

```
specific #$PCComponent, although in a specific PC configuration there
could be none, one or more expansion cards. Please, refer to the
#$PlugAndPlaySystem collection for more information.".

constant: PlugAndPlaySystem.
isa: Individual PCSubSystem.
comment: "It is the system responsible for loading the operating
system. In general, it comprises the #$FloppyDiskDrive with a floppy
disk containing the operating system (#$OSfloppyDisk) and the
#$HardDiskDrive, although a PC can be configured via the
#$BIOSsettings to use only one of them or both.".

constant: OSfloppyDisk.
isa: Individual PCComponent.
comment: "It is the floppy disk containing the operating system. It is
used by the #$BootSystem to load the OS from the #$FloppyDiskDrive.".

constant: BootSystem.
isa: Individual PCSubSystem.
comment: "It is the system responsible for loading the operating
system (OS). It includes the #$FloppyDiskDrive together with the floppy
disk containing the OS (#$OSfloppyDisk) and the #$HardDiskDrive. It
also includes the #$BootSequence-BIOSsetting which defines the sequence
in which these media will be used by the BIOS to load the OS.".


constant: functionalPartOf.
isa : TransitiveBinaryPredicate.
arg1Isa: PCSubSystem.
arg2Isa: PCSubSystem.
comment : "Predicate functionalPartOf is used to define a functional
          model of the PC under diagnosis
          functionalPartOf(WholeSubSystem PartialSubSystem) means
          that the PCSubSystem WholeSubSystem is using the function of
          PartialSubsystem to perforn its own function. E.g.,
          functionalPartOf(PowerSystem PowerSupply) means that for the
          PowerSystem to function the PowerSupply must function. This
          predicate is used to systematically disassemble the PC system into
          simpler PCSubSystems until a PCComponent is reached that it is
          faulty.".

constant: testFirst.
isa: UnaryPredicate.
arg1Isa: PCSubSystem.
comment: "This predicate is used to declare which PCSubSystem from
these occuring after a #$Decompose inference type will be the
the first to consider for diagnosis, i.e., the #$hypothesis.".

constant: testAfter.
isa: BinaryPredicate.
arg1Isa: PCSubSystem.
arg2Isa: PCSubSystem.
comment: "This predicate is used to declare the order of considering
PC subsystems for diagnosis. For example, (#$testAfter SUBSYSTEM1
SUBSYSTEM2) means that that the #$PCSubSystem SUBSYSTEM2 will
be considered for diagnosis immmediately after SUBSYSTEM1.".


Default Mt: PCDiagnosisMt.

constant: hypothesis.
isa: UnaryPredicate.
arg1Isa: PCSubSystem.
comment: "The predicate is used to record in the KB which
#$PCSubSystem is currently being diagnosed. E.g.,
#$hypothesis(#$VideoSystem) means that it is the #$VideoSystem that is
```

```
currently being checked for possible faults.".

constant: possibleHypotheses.
isa: UnaryPredicate.
arg1Isa: PCSubSystem.
comment: "The predicate is used to record in the KB which
#$PCSubSystems are currently candidates for being diagnosed. E.g.,
#$possibleHypotheses(#$VideoCard) means that the #$VideoCard is
a candidate to be checked for possible faults.".


;**************** SYSTEM TESTING KNOWLEDGE ****************

;;; The basic tool in the Systematic Diagnosis problem-solving method
;;; (PSM), as well as in any other diagnostic PSM, for carrying out
;;; the diagnostic procedure, is various TESTS that must be done to
;;; provide information (knowledge) about the state of the
;;; system. This knowledge may concern the actual behaviour of the
;;; system's components, e.g. the absence of electric power, control
;;; information produced by the system, e.g., beep codes or screen
;;; messages. Conceptually, a TEST is a question that the user must
;;; make to the system under diagnosis to extract knowledge about
;;; it. Here, it is implemented as a structure consisting of three
;;; other concepts:
;;; 1. The #$PCSubSystem to which it is related, i.e., to which the
;;;    question is adressed.
;;; 2. The #$TestAction which is the action one has really to perform
;;;    for the test
;;; 3. The #$PossibleObservable (system variable) that the TEST is
;;;    asking about .

;;; Although not part of a TEST structure, there is a fourth concept
;;; related with it, the  #$PossibleObservableValue (system variable
;;; value) which is the result (answer) of the TEST's question.

;;; Each TEST is represented as a non-atomic term (NAT) in CycL with
;;; the use of a #$NonPredicateFunction, #$TestFn, which takes as
;;; arguments instances of the three constituent concepts and returns
;;; a TEST structure, Schematically:

;;; (#$TestFN #$PCSubSystem #$TestAction #$PossibleObservable) -> #$Test

Default Mt: PCDiagnosisMt.

constant: Test.
isa: Collection.
genls: InformationBearingThing.
comment: "The basic tool in the Systematic Diagnosis problem-solving
method (PSM), as well as in any other diagnostic PSM, for carrying out
the diagnostic procedure, is various TESTS that must be done to
provide information (knowledge) about the state of the system. This
knowledge may concern the actual behaviour of the system's components,
e.g. the absence of electric power, control information produced by
the system, e.g., beep codes or screen messages. Conceptually, a TEST
is a question that the user must make to the system under diagnosis to
extract knowledge about it. Here, it is implemented as a structure
consisting of three other concepts: 1. The #$PCSubSystem to which it
is related, i.e., to which the question is adressed, 2. The
#$TestAction which is the action one has really to perform for the
test and 3. The #$PossibleObservable (system variable) that the TEST
is asking about.".


constant: TestFn.
isa: NonPredicateFunction.
arity: 3.
```

```
arg1Isa: PCSubSystem.
arg2Isa: TestAction.
arg3Isa: PossibleObservable.
resultIsa: Test.
comment: "Every #$Test is a structure consisting of three
concepts. The #$PCSubSystem to which it is related, the actual
#$TestAction that must be performed and the #$PossibleObservable
(system variable) that is being observed.Each TEST is represented as a
non-atomic term (NAT) in CycL
with the use of the #$NonPredicateFunction, #$TestFn, which takes as
arguments instances of the three constituent concepts and returns a
TEST structure, Schematically:
(#$TestFN #$PCSubSystem #$TestAction #$PossibleObservable) -> #$Test".

constant: possibleTest.
isa: UnaryPredicate.
arg1Isa: Test.
comment: "The #$Tests available to be performed in any stage of the
Diagnosis.".

constant: TestAction.
isa: Collection.
genls: PurposefulAction.
comment: "The collection of all possible test actions that may be
performed from the user on a #$PCSubSystem to determine the
#$PossibleObservableValues of
a #$PossibleObservable. These values are compared to the expected
ones. If they are different, the fault lies somewhere in the
#$PCSubSystem which is further decomposed and its functional parts are
checked one by one. If not, the fault lies in another #$PCSubSystem.".

constant: PossibleObservable.
isa: Collection.
genls: AttributeType.
comment: "The colection of system variables (possible observables)
which are used to decide if the currently checked #$PCSubSystem actually
contains a faulty #$PCComponent or not.".

constant: PossibleObservableValue.
isa: Collection.
genls: Thing.
comment: "The collection of all possible values of all
#$PossibleObservables. In terms of the Systematic Diagnosis
problem-solving method, the instances of #$PossibleObservable
correspond to the system variables that one can test during diagnosis and the
instances of #$PossibleObservableValue correspond to the possible
outcomes of these tests. These outcomes can be anything, therefore a
#$PossibleObservableValue is a sub-collection of #$Thing, the topmost
#$Collection ".

constant: ResultType.
isa: Collection.
genls: AttributeValue.
comment: "The collection of all various types of
#$PossibleObservableValues. These types are characterised from the
kind of conclusions they lead relative to the #$PCSubSystem being
currently diagnosed ( #$hypothesis(#$PCSubSystem) ). E.g., such a type
can be #$Normal which denotes that the #$PCSubSystem currently being
diagnosed is not faulty and therefore must be discarded as a
hypothesis and a new hypothesis must be selected.".

constant: possibleResultOfTest.
isa: Predicate.
arity: 3.
arg1Isa: Test.
arg2Isa: PossibleObservableValue.
```

```
arg3Isa: ResultType.
comment: "The predicate is correlating an  individual #$Test with its
actual result and the type of this result. The assertion
possibleResultOfTest(TEST VALUE TYPE)
express the fact that for the specific TEST, VALUE is a possible
result of type TYPE. E.g., possibleResultOfTest((TestFn
PowerSystem ConfirmSensorially ElectricPower) Yes Normal) indicates that
it is #$Normal to observe the existence of #$ElectricPower when
diagnosing the #$PowerSystem. Of course, this immediately would imply
that the #$PowerSystem is not faulty and therefore should be discarded
as a #$hypothesis.".

constant: resultOfTest.
isa: BinaryPredicate.
arg1Isa: Test.
arg2Isa: PossibleObservableValue.
comment: "The predicate is correlating an  individual #$Test with its
actual result. The assertion resultOfTest(?TEST ?VALUE) means that
?VALUE is the actual result of ?TEST.".

;******** DEFINITIONS OF INSTANCES FOR  #$TestAction COLLECTION ********

constant: ConfirmSensorially.
isa: TestAction.
comment: "The action of confirming the existence of a
#$PossibleObservable only by one's senses, e.g., visually,
acoustically.".


constant: CheckIndependently.
isa: TestAction.
comment: "This #$TestAction means that the user performing
diagnosis must check the function of the related #$PCSubSystem
isolated from the rest of the #$PCSubSystem of which it is a
#$functionalPartOf. The way to do that is not specifically described
by the name of the action. It is assumed that the user has some
knowledge for performing such isolated tests. E.g., to test the
#$PowerSocket one can plug another device - known to be working -  in
it and confirm that the device has #$ElectricPower.".

constant: Remove.
isa: TestAction.
comment: "This #$TestAction means that the user must remove the
related #$PCSubSystem from the #$PCSubSystem of which it is a
#$functionalPartOf".

constant: Replace.
isa: TestAction.
comment: "This #$TestAction means that the user must replace the
related #$PCSubSystem with a new one".

constant: ChangeVoltage.
isa: TestAction.
comment: "This #$TestAction means that the user must change the
voltage setting in the #$PowerSupply. It is a specific #$TestAction
related only to the #$PowerSupply".

constant: TroubleshootComponent.
isa: TestAction.
comment: "This #$TestAction means that the diagnosis reached at the
level of a specific PCComponent but there is not sufficient
information to confirm that it is faulty. Therefore, the user must
enter the stage of troubleshooting it specifically.".
```

```
;****** DEFINITIONS OF INSTANCES FOR  #$PossibleObservable COLLECTION ********

constant: ProblemContext.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the general context of
diagnosis, i.e., #$BootTime, #$RunTime, #$ComponentSpecific. The
value of this #$PossibleObservable determines which rules are
applicable, appearing as a condition in their antecedent part".

constant: ElectricPower.
isa: PossibleObservable.
comment: "The electric power that any #$PCSystem needs to operate.".

constant: VoltageCorrect.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the voltage setting in
the #$PowerSupply being correct, i.e., 110V or 220V.".

constant: VideoSignal.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the existence of any
video signal on the #$Monitor screen.".

constant: SpeakerBeep.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to any beep pattern coming
out of the #$Speaker.".

constant: VideoBIOSMessage.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the display of the
video BIOS message.".

constant: BootContinues.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the booting process
cointinuing normally.".

constant: StartupScreen.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the display of the BIOS
stratup screen.".

constant: MemoryTest.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the memeory test
performed by the BIOS during boot-time.".

constant: ErrorMessage.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the display of an error
message on the screen.".

constant: ComponentProblem.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the occasion where a
specific #$PCComponent has reached which is possibly faulty and the
only way to decide about this involves elaborate and complex #$Tests,
which the current implementation of the Systematic diagnosis problem
solving method does not cover. Therefore, the user has to perform
these #$Tests either based on his knowledge or have a human expert
perform them.".

constant: AutoDetection-BIOSsetting.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the #$HardDiskDrive
```

```
auto-detection setting in the PC BIOS. It may be set to #$Auto for
automatic detection or to #$Manual, usually the first one.".

constant: BootSequence-BIOSsetting.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the BIOS setting which
controls the sequence of the media used to load the operating
system. It may be A:-C: for using first the #$FloppyDiskDrive and then
the #$HardDiskDrive or C:-A: for the reverse.".

constant: BootSource.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the actual medium from
which the operating system is loaded, independently from what the
#$BootSequence-BIOSsetting is.".

constant: FloppyAccess.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to whether the
#$FloppyDiskDrive is actually accessed by the BIOS during boot-time
system test.".

constant: DetectionMessage.
isa: PossibleObservable.
comment: "This #$PossibleObservable is related to BIOS messages
concerning the autodetection of the #$HardDiskDrives.".

constant: InFloppy.
isa: PossibleObservable.
comment: "This #$PossibleObservable is related to the #$OSfloppyDisk
being inside the #$FloppyDiskDrive.".

;***** DEFINITIONS OF INSTANCES FOR $PossibleObservableValue COLLECTION ****

constant: BootTime.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed occured during boot-time, i.e., from the time the power is
turned on until the Operating System starts being loaded.".

constant: RunTime.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed occured during run-time, i.e., from the time the Operating
System starts being loaded until the PC is switched off.".

constant: ComponentSpecific.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed is identified to be related with a specific #$PCComponent,
e.g., the #$Monitor, #$MotherBoard, #$HardDisk e.t.c.".

constant: Yes.
isa: PossibleObservableValue.
comment: "Most of the #$Tests have as a possible result only 'Yes' or 'No'.".

constant: No.
isa: PossibleObservableValue.
comment: "Most of the #$Tests have as a possible result only 'Yes' or 'No'.".

constant: None.
isa: PossibleObservableValue.
comment: "This kind of result indicates that none of the alternative
```

```
results of a specific #$Test is observed.".

constant: SingleBeep.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$SpeakerBeep #$PossibleObservable. It means that the #$Speaker
produced a single beep".

constant: RingingOrBuzzing.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$SpeakerBeep #$PossibleObservable. It means that the #$Speaker is
producing a ringing or buzzing sound.".

constant: ConsistentPattern.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$SpeakerBeep #$PossibleObservable. It means that the #$Speaker is
producing a consistent pattern (code) of beeps, e.g., one beep, then
two more.".

constant: Complete.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to some tests
performed by the BIOS, e.g., the memory test. It means that the
corresponding test is succesfully completed.".

constant: InComplete.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to some tests
performed by the BIOS, e.g., the memory test. It means that the
corresponding test is not succesfully completed.".

constant: CannotFind-Message.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to BIOS error
messages concerning the autodetection of IDE/ATAPI devices,
e.g. #$HardDiskDrive, #$CDROMdrive. This kind of messages indicate
that the BIOS cannot detect the corresponding device.".


;"This #$PossibleObservableValue is related to BIOS error messages
;concerning the autodetection of the #$HardDiskDrives. This kind of
;messages indicate that the BIOS cannot detect any #$HardDiskDrive.".

constant: Auto.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue indicates that some
action/process/procedure is (set to be) done automatically.".

constant: Manual.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue indicates that some
action/process/procedure is (set to be) done manually.".

constant: FloppyThenHard.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$BootSequence-BIOSsetting #$PossibleObservable. It indicates that
this setting is set to A:-C:.".

constant: HardThenFloppy.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$BootSequence-BIOSsetting #$PossibleObservable. It indicates that
this setting is set to C:-A:.".
```

```
;***** DEFINITIONS OF INSTANCES FOR $ResultType COLLECTION ****

constant: Normal.
isa: ResultType.
comment: "This type of result indicates that the result is normally
expected when the #$PCSubSystem related with it is working
properly. Such a kind of result implies that the #$PCSubSystem must be
discarded as a #$hypothesis.".

constant: NotNormal.
isa: ResultType.
comment: "This type of result indicates that the result is not normally
expected when the #$PCSubSystem related with it is working
properly. Such a kind of result implies that the fault lies in the
#$PCSubSystem which is the current #$hypothesis.".

constant: Insufficient.
isa: ResultType.
comment: "This type of result indicates that the result cannot
undoubtedly indicate either the normal function or the malfunction of
the #$PCComponent related with it. Such a kind of result implies that
further testing is necessary to decide about the functional status of
the #$PCComponent which is the current #$hypothesis.".

constant: Distinguishing.
isa: ResultType.
comment: "This type of result occurs in a situation where there are
two components that are probably faulty and the only way to find
which, is to test one of them. In this case, a result of type
#$Distinguishing indicates simultaneously two things. First, that the
#$PCSubSystem hypothesised as faulty is not such and second, that the faulty
one is the other alternative #$PCSubSystem.".


;;; ********* IMPLEMENTATION OF KADS SYSTEMATIC DIAGNOSIS PSM ********

;;; The Task Structure for Systematic Diagnosis (pseudo-code) is:

;;; Systematic Diagnosis(+complaint,+possible observables,-hypothesis) by
;;;  select1(+complaint, -system model)
;;;  REPEAT
;;;   decompose(+system model, -hypothesis)
;;;   WHILE number of hypotheses > 1
;;;    select2(+possible observables, -variable value)
;;;    select3(+hypothesis, -norm)
;;;    compare(+variable value, +norm, -difference)
;;;   system model <- current decomposition level of system model
;;;  UNTIL confirm(+hypothesis), i.e. system model cannot be decomposed further

;;; The user interaction in CYC will be done from the SubL Interactor
;;; interface, as it is not possible to get any input/output
;;; interaction between the user and the SubL code from the ASK
;;; interface. The whole Task Structure will be implemented as a SubL
;;; function, 'systematic', which will be responsible for calling the
;;; appropriate SubL functions that will implement the corresponding
;;; inferences. In fact, the 'select1', 'select2', and 'select3'
;;; inferences will be implemented as FORWARD rules in the
;;; CYC KB. "Forward" means that, according to the results of 'Tests'
;;; that the user is asked to give, these rules automatically assert
;;; new facts in the KB. These facts describe which are the next
;;; #$PossibleObservables and Variables that must be tested ('select2'
```

```
;;; inference), what should be done according to the result ('select3'
;;; and 'compare' inferences), e.g., if another test for the same
;;; hypothesis should be performed or if the current hypothesis should
;;; be rejected or the current hypothesis must be decomposed further
;;; or if the faulty component was found ('confirm' inference).


;;; The following three (3) constant definitions introduce the
;;; #$plausibleInference predicate and #$Decompose inference type of
;;; KADS. These two are used in the antecendent part of the
;;; "decomposition" rules. They do not constitute control knowledge
;;; but domain role knowledge. In terms of implementation, they cause
;;; the forward "decomposition" rules to fire only when a Decompose
;;; inference has to be made.

Default Mt: PCDiagnosisMt.

constant: InferenceType-KADS.
isa: Collection.
genls:  PropositionalInformationThing.
comment: "The collection of all Inference Types of KADS methodology,
e.g., 'select', 'decompose', 'confirm'.".

constant: Decompose.
isa: InferenceType-KADS.
comment: "The #$Decompose inference type of KADS takes a structured
hierarchy of objects and gives a less or completely unstructured
collection of these objects. In its simplest form it is used for
breaking down existing knowledge structures, like hierarchies, where
there is no loss of objects but only the structure is removed.".

constant: plausibleInference.
isa: UnaryPredicate.
arg1Isa: InferenceType-KADS.
comment: "The #$plausibleInference predicate is used to record in the
KB which inference(s) can be next performed during the 'execution' of
a problem solving method.".


;;; During the Systematic Diagnosis problem solving method (PSM) as
;;; well as during any other PSM, there are certain decisions/choices
;;; that must be done. According to the structure of the PSMs as
;;; Generic Task Models in KADS, these decisions/choices occur during
;;; the performance of specific Inferences. The implementation of
;;; these decisions/choices has to be declarative since this is the
;;; main principle in CYC. Therefore, the implementation of them will
;;; be in terms of FORWARD rules: the ANTECENDENT of each rule will be
;;; the conditions under which a decision is made ant the CONSEQUENT
;;; will be the knowledge that is becoming known to the system when
;;; this decision is made. Then, the newly added information will be
;;; used by the SubL code to guide the whole procedure. In the
;;; following, we will examine in detail which these decisions/choices
;;; are, when and where do they occur and how the are actually
;;; implemente as forward rules.

;;; The PSM starts with a SELECT inference. A general symptom is
;;; entered by the user and an appropriate system model is
;;; chosen. This inference is slightly changed for PC diagnosis. What
;;; is actually asked from the user is to distinguish three (3) major
;;; contexts of diagnosis: (i) Boot-time troubleshooting, (ii)
;;; Run-time troubleshooting and (iii) Component-specific
;;; troubleshooting. This categorisation is significant since
;;; completely different rules are applicable in each
;;; context. Although this contextual dependency of the rules could be
;;; implemented as different #$Microtheory contexts, this would make
;;; context shifting more complicated - any ASK operation would have
```

```
;;; to define the #$Microtheory. Instead, this dependency will be
;;; embedded in the antecedent part of the rules as an extra
;;; condition. The special predicate diagnosisContext will be used, e.g.,
;;;
;;; (implies
;;;  (and
;;;   (diagnosisContext BootTime)
;;;   (...<more conditions>...)) ;end of antecedent
;;;  (<consequent>))

constant: diagnosisContext.
isa: UnaryPredicate.
arg1Isa: PossibleObservableValue.
comment: "This predicate records the current problem-solving context,
i.e. #$BootTime, #$RunTime or #$ComponentSpecific.".

;;; Rule to assert a (#$diagnosisContext ...) assertion. This assertion
;;; is introduced as a "shorthand" for the assertion:
;;;
;;; (resultOfTest (TestFn PCSystem ConfirmSensorially ProblemContext) ?PROBLEM)
;;;
;;; It is used as a premise in every rule which is applicable to the
;;; corresponding diagnosis context, i.e., #$BootTime, #$RunTime or
;;; #$ComponentSecific.

Default Mt: PCDiagnosisMt.

Direction: forward.
F: (implies
    (resultOfTest (TestFn PCSystem ConfirmSensorially ProblemContext) ?PROBLEM)
    (diagnosisContext ?PROBLEM)).

;;; The 'decompose' inference in KADS Systematic Diagnosis PSM takes as input
;;; the current #$PCSubSystem (#$hypothesis PC_SUBSYSTEM) and decomposes it
;;; into its functional subsystems (#$functionalPartOf PC_SUBSYSTEM PART),
;;; generating new hypotheses (#$possibleHypotheses PART).

;;; **********************************
;;; ** Rules and Facts for Decompose **
;;; **********************************

;; Every PART which is a functional part of the hypothesis, HYP, is a
;; possible hypothesis.

direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (plausibleInference Decompose)
     (hypothesis ?HYP)
     (functionalPartOf ?HYP ?PART))
    (possibleHypotheses ?PART)).


F: (functionalPartOf PCSystem PowerSystem).
F: (functionalPartOf PCSystem VideoSystem).
F: (functionalPartOf PCSystem BIOSStartupSystem).
F: (functionalPartOf PCSystem MemorySystem).
F: (functionalPartOf PCSystem FloppySystem).
F: (functionalPartOf PCSystem HardDiskDrive).
F: (functionalPartOf PCSystem CDROMdrive).
F: (functionalPartOf PCSystem PlugAndPlaySystem).
F: (functionalPartOf PCSystem BootSystem).
```

```
F: (functionalPartOf PowerSystem PowerSocket).
F: (functionalPartOf PowerSystem PowerCable).
F: (functionalPartOf PowerSystem PowerProtectionDevice).
F: (functionalPartOf PowerSystem PowerSupply).


F: (functionalPartOf VideoSystem MotherBoard).
F: (functionalPartOf VideoSystem VideoCard).
F: (functionalPartOf VideoSystem Monitor).

F: (functionalPartOf BIOSStartupSystem MotherBoard).
F: (functionalPartOf BIOSStartupSystem VideoCard).

F: (functionalPartOf MemorySystem RAM).
F: (functionalPartOf MemorySystem MotherBoard).

F: (functionalPartOf FloppySystem FloppyDiskDrive).
F: (functionalPartOf FloppySystem BIOSsettings).


F: (functionalPartOf PlugAndPlaySystem ExpansionCard).
F: (functionalPartOf PlugAndPlaySystem MotherBoard).

F: (functionalPartOf BootSystem FloppyDiskDrive).
F: (functionalPartOf BootSystem OSfloppyDisk).
F: (functionalPartOf BootSystem HardDiskDrive).

;;; ********************************************************
;;; **                                                    **
;;; ** Facts and rules about PC subsystems and components **
;;; **                                                    **
;;; ********************************************************

;;; ******************************
;;; ** TEST(S) for the PCSystem **
;;; ******************************

Default Mt: PCDiagnosisMt.

Direction: forward.
F: (implies
    (hypothesis PCSystem) ;if diagnosis just started, ask for the context
    (and
     (possibleTest (TestFn PCSystem ConfirmSensorially ProblemContext))
     (possibleResultOfTest
      (TestFn PCSystem ConfirmSensorially ProblemContext) BootTime NotNormal)
     (possibleResultOfTest
      (TestFn PCSystem ConfirmSensorially ProblemContext) RunTime  NotNormal)
     (possibleResultOfTest
      (TestFn PCSystem ConfirmSensorially ProblemContext)
                                               ComponentSpecific NotNormal))).

;; **********************************************
;; ** DECOMPOSITION knowledge for the PCSystem **
;; **********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PCSystem)
     (plausibleInference Decompose))
    (and
     (testFirst PowerSystem)
```

```
     (testAfter PowerSystem VideoSystem)
     (testAfter VideoSystem BIOSStartupSystem)
     (testAfter BIOSStartupSystem MemorySystem)
     (testAfter MemorySystem FloppySystem)
     (testAfter FloppySystem HardDiskDrive)
     (testAfter HardDiskDrive CDROMdrive)
     (testAfter CDROMdrive PlugAndPlaySystem)
     (testAfter PlugAndPlaySystem BootSystem))).


;; ********************************
;; ** TEST(S) for the PowerSystem **
;; ********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSystem)) ;
    (and
     (possibleTest (TestFn PowerSystem ConfirmSensorially ElectricPower))
     (possibleResultOfTest
      (TestFn PowerSystem ConfirmSensorially ElectricPower) Yes Normal)
     (possibleResultOfTest
      (TestFn PowerSystem ConfirmSensorially ElectricPower) No NotNormal))).

;; ************************************************
;; ** DECOMPOSITION knowledge for the PowerSystem **
;; ** ElectricPower=No                            **
;; ************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSystem)
     (resultOfTest (TestFn PowerSystem ConfirmSensorially ElectricPower) No)
     (plausibleInference Decompose))
    (and
     (testFirst PowerSocket)
     (testAfter PowerSocket PowerProtectionDevice)
     (testAfter PowerProtectionDevice PowerCable)
     (testAfter PowerCable PowerSupply))).

;; ********************************
;; ** TEST(S) for the PowerSocket **
;; ********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSocket)) ;
    (and
     (possibleTest (TestFn PowerSocket CheckIndependently ElectricPower))
     (possibleResultOfTest
      (TestFn PowerSocket CheckIndependently ElectricPower) Yes Normal)
     (possibleResultOfTest
      (TestFn PowerSocket CheckIndependently ElectricPower) No NotNormal))).

;;******************************************
;; ** TEST(S) for the PowerProtectionDevice **
;;******************************************

Direction: forward.
```

```
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerProtectionDevice))
    (and
     (possibleTest (TestFn PowerProtectionDevice Remove ElectricPower))
     (possibleResultOfTest
      (TestFn PowerProtectionDevice Remove ElectricPower) Yes NotNormal)
     (possibleResultOfTest
      (TestFn PowerProtectionDevice Remove ElectricPower) No Normal))).


;;*********************************
;; ** TEST(S) for the PowerCable **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerCable)) ;
    (and
     (possibleTest (TestFn PowerCable Replace ElectricPower))
     (possibleResultOfTest
      (TestFn PowerCable Replace ElectricPower) Yes NotNormal)
     (possibleResultOfTest
      (TestFn PowerCable Replace ElectricPower) No Normal))).

;;*********************************
;;** TEST(S) for the PowerSupply **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSupply))
    (and
     (possibleTest (TestFn PowerSupply ConfirmSensorially VoltageCorrect))
     (possibleResultOfTest
      (TestFn PowerSupply ConfirmSensorially VoltageCorrect) Yes NotNormal)
     (possibleResultOfTest
      (TestFn PowerSupply ConfirmSensorially VoltageCorrect) No Insufficient))).

;;*********************************
;;** TEST(S) for the PowerSupply **
;;** VoltageCorrect=No          **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSupply)
     (resultOfTest (TestFn PowerSupply ConfirmSensorially VoltageCorrect) No)) ;
    (and
     (possibleTest (TestFn PowerSupply ChangeVoltage ElectricPower))
     (possibleResultOfTest
      (TestFn PowerSupply ChangeVoltage ElectricPower) Yes Normal)
     (possibleResultOfTest
      (TestFn PowerSupply ChangeVoltage ElectricPower) No NotNormal))).

;;**********************************
;;** TEST(S) for the VideoSystem **
;;**********************************
```

```
Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem))
    (and
     (possibleTest (TestFn VideoSystem ConfirmSensorially VideoSignal))
     (possibleResultOfTest
      (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes Insufficient)
     (possibleResultOfTest
      (TestFn VideoSystem ConfirmSensorially VideoSignal) No NotNormal))).

;; ************************************************
;; ** DECOMPOSITION knowledge for the VideoSystem **
;; ** VideoSignal=No                             **
;; ************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No)
     (plausibleInference Decompose))
    (and
     (testFirst Monitor)
     (testAfter Monitor MotherBoard)
     (testAfter MotherBoard Speaker))).

;;*********************************
;;** TEST(S) for the VideoSystem **
;;** VideoSignal=Yes             **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes))
    (and
     (possibleTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage))
    (possibleResultOfTest
     (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes Insufficient)
    (possibleResultOfTest
     (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No NotNormal))).

;; ******************************************
;; ** TEST(S) for the VideoSystem          **
;; ** VideoSignal=Yes, VideoBIOSMessage=Yes **
;; ******************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes))
    (and
     (possibleTest (TestFn VideoSystem ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn VideoSystem ConfirmSensorially BootContinues) Yes Normal)
     (possibleResultOfTest
      (TestFn VideoSystem ConfirmSensorially BootContinues) No NotNormal))).
```

```
;; *************************************************************
;; ** DECOMPOSITION knowledge for the VideoSystem             **
;; ** VideoSignal=Yes, VideoBIOSMessage=Yes, BootContinues=No **
;; *************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially BootContinues) No)
     (plausibleInference Decompose))
    (and
     (testFirst VideoCard)
     (testAfter VideoCard MotherBoard))).

;; **************************************************
;; ** DECOMPOSITION knowledge for the VideoSystem  **
;; ** VideoSignal=Yes, VideoBIOSMessage=No         **
;; **************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
     (plausibleInference Decompose))
    (and
     (testFirst MotherBoard)
     (testAfter MotherBoard VideoCard))).

;; *****************************
;; ** TEST(S) for the Monitor **
;; ** VideoSignal=No          **
;; *****************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis Monitor)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No))
    (and
     (possibleTest (TestFn Monitor CheckIndependently VideoSignal))
     (possibleResultOfTest
      (TestFn Monitor CheckIndependently VideoSignal) Yes Normal)
     (possibleResultOfTest
      (TestFn Monitor CheckIndependently VideoSignal) No NotNormal))).

;; ******************************************
;; ** TEST(S) for the MotherBoard          **
;; ** VideoSignal=No, Monitor_Working =Yes **
;; ******************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
```

```
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No)
     (resultOfTest (TestFn Monitor CheckIndependently VideoSignal) Yes))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) SingleBeep NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep)
                                        ConsistentPattern NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep)
                                        RingingOrBuzzing NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient))).

;; ********************************************************
;; ** TEST(S) for the MotherBoard                        **
;; ** VideoSignal=No, Monitor_Working =Yes, SpeakerBeep=No **
;; ********************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No)
     (resultOfTest (TestFn Monitor CheckIndependently VideoSignal) Yes)
     (resultOfTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No))
    (and
     (possibleTest (TestFn Speaker CheckIndependently SpeakerBeep))
     (possibleResultOfTest
      (TestFn Speaker CheckIndependently SpeakerBeep) Yes NotNormal)
     (possibleResultOfTest
      (TestFn Speaker CheckIndependently SpeakerBeep) No Distinguishing))).

;;; Important notice: Although the PCSusbSystem related to the Test is
;;; the Speaker, however, the PCSubSystem hypothesised as faulty is
;;; the MotherBoard. Therefore, the Result Type is defined by what it
;;; indicates about the MotherBoard and not the Speaker itself. This
;;; rather peculiar situation is due to the controlling function of
;;; the Speaker. This means that the Speaker functions as a control
;;; device for the MotherBoard and we must make sure that this device
;;; is working properly. If it does, then the lack of any sound is due
;;; to the MotherBoard and we can deduce that it is faulty. Otherwise,
;;; we cannot deduce anything before we are certain that the Speaker
;;; is working properly.


;; ****************************************
;; ** TEST(S) for the MotherBoard        **
;; ** VideoSignal=Yes, VideoBIOSMessage=No **
;; ****************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) ConsistentPattern
```

```
                                                               Insufficient))).

;; ************************************************************************
;; ** TEST(S) for the MotherBoard                                        **
;; ** VideoSignal=Yes, VideoBIOSMessage=No, SpeakerBeep=ConsistentPattern **
;; ************************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
     (resultOfTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep)
                                                            ConsistentPattern))
    (and
     (possibleTest (TestFn MotherBoard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; *********************************************************
;; ** TEST(S) for the MotherBoard                          **
;; ** VideoSignal=Yes, VideoBIOSMessage=No, SpeakerBeep=No **
;; *********************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
     (resultOfTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No))
    (and
     (possibleTest (TestFn MotherBoard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; *********************************
;; ** TEST(S) for the MotherBoard **
;; ** StartupScreen=No            **
;; *********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest
       (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) Yes NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient))).
```

```
;; *************************************
;; ** TEST(S) for the MotherBoard      **
;; ** StartupScreen=No, SpeakerBeep=No **
;; *************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest
      (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No)
     (resultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially ErrorMessage))
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially ErrorMessage) Yes NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially ErrorMessage) No Insufficient))).

;; ****************************************************
;; ** TEST(S) for the MotherBoard                    **
;; ** StartupScreen=No, SpeakerBeep=No, ErrorMessage=No **
;; ****************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest
      (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No)
     (resultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No)
     (resultOfTest
      (TestFn MotherBoard ConfirmSensorially ErrorMessage) No))
    (and
     (possibleTest (TestFn MotherBoard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) Yes
                                                                  NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) No
                                                          Distinguishing))).



;; ***********************************************************
;; ** TEST(S) for the VideoCard                             **
;; ** VideoSignal=Yes, VideoBIOSMessage=Yes, BootContinues=No **
;; ***********************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoCard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially BootContinues) No))
    (and
     (possibleTest (TestFn VideoCard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn VideoCard TroubleshootComponent ComponentProblem) Yes NotNormal)
     (possibleResultOfTest
```

```
     (TestFn VideoCard TroubleshootComponent ComponentProblem) No
                                                        Distinguishing))).

;;; Important notice: The '#$TroubleshootComponent' TestAction and the
;;; '#$ComponentProblem' PossibleObservable are too general and
;;; complex to be of actual use. They are used here as artificial
;;; "terminating points" of the Systematic Diagnosis procedure at the
;;; level of #$PCComponent. In a
;;; complete PC faults diagnosis expert system, more elaborate
;;; 'Test(s)' should follow to troubleshoot a specific #$PCComponent
;;; (here the #$VideoCard), instead of the user having to know how to
;;; test the specific #$PCComponent. However, these 'Tests' are so
;;; elaborate and complicated that are beyond the scope of this
;;; implementation. Such 'Test(s)' could be identifying a beep code
;;; according to the specific version of BIOS (American Megatrends
;;; Inc., Pheonix or Other) or interpreting an error message (there
;;; are 120 error messages documented in the PCGuide Troubleshoot
;;; expert that was used as a knowledge acquisistion source).

;; *************************************
;; ** TEST(S) for the BIOSStartupSystem **
;; ** PowerSystem=ok, VideoSystem=ok    **
;; *************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BIOSStartupSystem))
    (and
     (possibleTest (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen))
     (possibleResultOfTest
      (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) Yes Normal)
     (possibleResultOfTest
     (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No NotNormal))).

;; ****************************************************
;; ** DECOMPOSITION knowledge for the BIOSStartupSystem **
;; ** StartupScreen=No                                 **
;; ****************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BIOSStartupSystem)
     (resultOfTest
       (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No)
     (plausibleInference Decompose))
    (and
     (testFirst MotherBoard)
     (testAfter MotherBoard VideoCard))).

;; ***************************************************
;; ** TEST(S) for the MemorySystem                   **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ***************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MemorySystem))
    (and
     (possibleTest (TestFn MemorySystem ConfirmSensorially MemoryTest))
     (possibleResultOfTest
```

```
      (TestFn MemorySystem ConfirmSensorially MemoryTest) Complete Normal)
     (possibleResultOfTest
     (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete NotNormal))).

;; **************************************************
;; ** DECOMPOSITION knowledge for the MemorySystem **
;; ** MemoryTest=Incomplete                        **
;; **************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MemorySystem)
     (resultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete)
     (plausibleInference Decompose))
    (and
     (testFirst RAM)
     (testAfter RAM MotherBoard))).

;; ***************************
;; ** TEST(S) for the RAM   **
;; ** MemoryTest=Incomplete **
;; ***************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis RAM)
     (resultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete))
    (and
     (possibleTest (TestFn RAM ConfirmSensorially ErrorMessage))
     (possibleResultOfTest
      (TestFn RAM ConfirmSensorially ErrorMessage) Yes NotNormal)
     (possibleResultOfTest
      (TestFn RAM ConfirmSensorially ErrorMessage) No Insufficient))).

;; ********************************************
;; ** TEST(S) for the RAM                    **
;; ** MemoryTest=Incomplete, ErrorMessage=No **
;; ********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis RAM)
     (resultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete)
     (resultOfTest
      (TestFn RAM ConfirmSensorially ErrorMessage) No))
    (and
     (possibleTest (TestFn RAM TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn RAM TroubleshootComponent ComponentProblem) Yes NotNormal)
     (possibleResultOfTest
      (TestFn RAM TroubleshootComponent ComponentProblem) No
                                                            Distinguishing))).

;; ******************************************************
;; ** TEST(S) for the FloppySystem                     **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok                                  **
```

```
;; ****************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis FloppySystem))
    (and
     (possibleTest (TestFn FloppySystem ConfirmSensorially FloppyAccess))
     (possibleResultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) Yes Normal)
     (possibleResultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) No NotNormal))).

;; ************************************************
;; ** DECOMPOSITION knowledge for the FloppySystem **
;; ** FloppyAccess=No                            **
;; ************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis FloppySystem)
     (resultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) No)
     (plausibleInference Decompose))
    (and
     (testFirst FloppyDiskDrive)
     (testAfter FloppyDiskDrive BIOSsettings))).

;; ***********************************
;; ** TEST(S) for the FloppyDiskDrive **
;; ** FloppyAccess=No               **
;; ***********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis FloppyDiskDrive)
     (resultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) No))
    (and
     (possibleTest (TestFn FloppyDiskDrive ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn FloppyDiskDrive ConfirmSensorially BootContinues) Yes
                                                              Distinguishing)
     (possibleResultOfTest
      (TestFn FloppyDiskDrive ConfirmSensorially BootContinues) No NotNormal))).


;; *****************************************************************
;; ** TEST(S) for the OSfloppyDisk                                **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=Hard/None **
;; *****************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis OSfloppyDisk)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                               FloppyThenHard)
     (or
```

```
       (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource)
                                                                 HardDiskDrive)
       (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource) None)))
    (and
     (possibleTest (TestFn OSfloppyDisk ConfirmSensorially InFloppy))
     (possibleResultOfTest
      (TestFn OSfloppyDisk ConfirmSensorially InFloppy) Yes Insufficient)
     (possibleResultOfTest
      (TestFn OSfloppyDisk ConfirmSensorially InFloppy) No NotNormal))).

;; *******************************************************************
;; ** TEST(S) for the OSfloppyDisk                                  **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=Hard/None **
;; ** InFloppy=Yes                                                  **
;; *******************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis OSfloppyDisk)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
     (or
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource)
                                                                 HardDiskDrive)
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource) None))
     (resultOfTest
      (TestFn OSfloppyDisk ConfirmSensorially InFloppy) Yes))
    (and
     (possibleTest
      (TestFn OSfloppyDisk TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn OSfloppyDisk TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn OSfloppyDisk TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; ******************************************************
;; ** TEST(S) for the HardDiskDrive                    **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok, FloppySystem=ok                 **
;; ******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis HardDiskDrive))
    (and
     (possibleTest (TestFn HardDiskDrive ConfirmSensorially DetectionMessage))
     (possibleResultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage) Yes Normal)
     (possibleResultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage) No Insufficient)
     (possibleResultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage)
                                               CannotFind-Message NotNormal))).

;; ***********************************
;; ** TEST(S) for the HardDiskDrive **
;; ** DetectionMessage=No           **
;; ***********************************
```

```
Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis HardDiskDrive)
     (resultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage) No))
    (and
     (possibleTest
      (TestFn HardDiskDrive CheckIndependently AutoDetection-BIOSsetting))
     (possibleResultOfTest
      (TestFn HardDiskDrive CheckIndependently AutoDetection-BIOSsetting)
                                                                  Manual Normal)
     (possibleResultOfTest
     (TestFn HardDiskDrive CheckIndependently AutoDetection-BIOSsetting)
                                                              Auto NotNormal))).

;; *********************************************************************
;; ** TEST(S) for the HardDiskDrive                                   **
;; ** BootSequence-BIOSsetting=HardThenFloppy, BootSource=Floppy/None **
;; *********************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis HardDiskDrive)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                 HardThenFloppy)
     (or
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource)
                                                                FloppyDiskDrive)
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource) None)))
    (and
     (possibleTest
      (TestFn HardDiskDrive TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn HardDiskDrive TroubleshootComponent ComponentProblem) Yes
                                                                      NotNormal)
     (possibleResultOfTest
      (TestFn HardDiskDrive TroubleshootComponent ComponentProblem) No
                                                              Normal))).

;; ******************************************************
;; ** TEST(S) for the CDROMdrive                       **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok, FloppySystem=ok, HardDisk=ok    **
;; ******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis CDROMdrive))
    (and
     (possibleTest (TestFn CDROMdrive ConfirmSensorially DetectionMessage))
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) Yes Normal)
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) No Insufficient)
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) CannotFind-Message
                                                                   NotNormal))).

;; ***********************************
```

```
;; ** TEST(S) for the CDROMdrive **
;; ** DetectionMessage=No           **
;; **********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis CDROMdrive)
     (resultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) No))
    (and
     (possibleTest (TestFn CDROMdrive ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially BootContinues) Yes Normal)
     (possibleResultOfTest
     (TestFn CDROMdrive ConfirmSensorially BootContinues) No NotNormal))).

;; *****************************************************
;; ** TEST(S) for the PlugAndPlaySystem               **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok, FloppySystem=ok, HardDisk=ok    **
;; ** CDROMdrive=ok                                    **
;; *****************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PlugAndPlaySystem))
    (and
     (possibleTest (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) Yes Normal)
     (possibleResultOfTest
     (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) No NotNormal))).

;; ******************************************************
;; ** DECOMPOSITION knowledge for the PlugAndPlaySystem **
;; ** BootContinues=No                                  **
;; ******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PlugAndPlaySystem)
     (resultOfTest
      (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) No)
     (plausibleInference Decompose))
    (and
     (testFirst ExpansionCard)
     (testAfter ExpansionCard MotherBoard))).

;; **********************************
;; ** TEST(S) for the ExpansionCard **
;; ** BootContinues=No             **
;; **********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis ExpansionCard)
     (resultOfTest
      (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) No))
```

```
    (and
     (possibleTest
      (TestFn ExpansionCard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn ExpansionCard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn ExpansionCard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; **********************************
;; ** TEST(S) for the BootSystem   **
;; ** Everything else=ok          **
;; **********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem))
    (and
     (possibleTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting))
     (possibleResultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                   FloppyThenHard Insufficient)
     (possibleResultOfTest
     (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                HardThenFloppy Insufficient))).

;; **********************************************
;; ** TEST(S) for the BootSystem               **
;; ** BootSequence-BIOSsetting=FloppyThenHard  **
;; **********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                               FloppyThenHard))
    (and
     (possibleTest (TestFn BootSystem ConfirmSensorially BootSource))
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) FloppyDiskDrive Normal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                                       HardDiskDrive NotNormal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None NotNormal))).

;; **************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem               **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=Hard **
;; **************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
```

```
     (resultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) HardDiskDrive)
     (plausibleInference Decompose))
    (and
     (testFirst OSfloppyDisk)
     (testAfter  OSfloppyDisk FloppyDiskDrive))).


;; **************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem               **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=None **
;; **************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
     (resultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None)
     (plausibleInference Decompose))
    (and
     (testFirst OSfloppyDisk)
     (testAfter OSfloppyDisk FloppyDiskDrive))).

;; **********************************************
;; ** TEST(S) for the BootSystem               **
;; ** BootSequence-BIOSsetting=HardThenFloppy  **
;; **********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                               HardThenFloppy))
    (and
     (possibleTest (TestFn BootSystem ConfirmSensorially BootSource))
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) HardDiskDrive Normal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                                     FloppyDiskDrive NotNormal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None NotNormal))).

;; ****************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem                 **
;; ** BootSequence-BIOSsetting=HardThenFloppy, BootSource=Floppy **
;; ****************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                HardThenFloppy)
     (resultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) FloppyDiskDrive)
```

```
     (plausibleInference Decompose))
   (testFirst HardDiskDrive)).


;; ************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem             **
;; ** BootSequence-BIOSsetting=HardThenFloppy, BootSource=None **
;; ************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                             HardThenFloppy)
     (resultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None)
     (plausibleInference Decompose))
    (testFirst HardDiskDrive)).
```

Appendix C

The CYC SubL Code for PC Domain

```
;;; ******** FUNCTION DEFINITIONS ********

;;; ******** GLOBAL VARIABLES ****************

(csetq *use-local-queue?* NIL) ;CYC variable
(defvar *defaultMt* '#$PCDiagnosisMt) ;the microtheory for the fi-ask function
(defvar *test* nil) ; the current Test (needed by the 'menu' function)
(defvar *results* nil) ; a list of the Possible Results of the current Test
                       ; (needed by the 'menu' function)
(defvar *no_of_choices* 0) ;the number of Possible Results of the current Test
                           ; (needed by the 'menu' function)

;;; The global variable *terms* is used to record in a list - and in
;;; parallel with what the 'format' expressions print to the CYC SubL
;;; Interactor panel - the CYC terms involved in each step of the
;;; Systematic diagnosis. This variable is returned to the Interactor
;;; panel as the 'Results' of the 'Last Form Evaluated' (see the CYC
;;; SubL Interactor panel). The Interactor panel converts any CYC
;;; term, i.e. symbols starting with '#$' (hash-dollar), as an HTML
;;; link to this term in the CYC KB. This way, the user can browse to
;;; any of the CYC terms appearing on the screen.

(defvar *terms* nil)



;;; *********************************************
;;; ******** SubL CODE FOR SYSTEMATIC ************
;;; *********************************************

;;; Function : SYSTEMATIC
;;; Arguments: The system that is going to be diagnosed, i.e. '#$PCSystem'.
;;; Result   : Implements the Inference Structure for the Systematic
;;;            Diagnosis problem solving method of KADS applied in PC
;;;            faults diagnosis.
;;; Remarks  :

(define systematic (system)
  (pcond
  ((cnot (eql system '#$PCSystem)) (error "DIAGNOSIS: argument must be
                                                          '#$PCSystem'"))
  (t (pcond
      ;; if there isn't any hypothesis then start diagnosis
      ((cnot (fi-ask '(#$hypothesis ?HYP) *defaultMt*))
```

```
        (csetf *terms* nil)
        (sd-select1))
      (t (sd-compare)) ;end of innermost 't' clause
     )) ;end of innermost 'pcond' and outermost 't' clause
 ) ;end of outermost 'pcond'
)

;;; ********************************************
;;; ******** SubL CODE FOR SD-COMPARE * *************
;;; ********************************************

;;; Function : SD-COMPARE
;;; Arguments: None
;;; Result   : Calls the appropriate Systematic Diagnosis Inference
;;; Remarks  : The *test* global variable holds the last 'Test' that
;;;            was performed. It is used to retrieve the result of
;;;            this 'Test' and its type. According to the 'ResultType'
;;;            of the 'Test', the following decisions may be made:
;;;
;;;            RESULT_TYPE            DECISION
;;;
;;;             Normal                 Reject hypothesis; backtrack
;;;             NotNormal              Decompose/Confirm hypothesis
;;;             Insufficient           Perform another 'Test' (Select2-3)
;;;             Distinguishing         Reject hypothesis, backtrack
;;;                                    and confirm next hypothesis.

(define sd-compare ()
 (csetf *terms* nil)
;; get the 'ResultType' for the most recently performed 'Test'
 (csetq ask-result
   (fi-ask (list '#$and
            (list '#$resultOfTest *test* '?RES)
            (list '#$possibleResultOfTest *test* '?RES '?TYPE)) *defaultMt*))
 (format t "LAST TEST: ~S,      ~%RESULT: ~S," *test* (get-ask-binding
              (first ask-result) 1))
 (csetf *terms* (cons (cons 'LAST (cons 'TEST: *test*)) *terms*))

 (csetq result (get-ask-binding (first ask-result) 1))
 (csetq result-type (get-ask-binding (first ask-result) 2))

;; according to the 'ResultType' value, decide the next inference
 (pcond
  ((eql result-type '#$NotNormal)
     (format t " RESULT TYPE: #$NotNormal, INFERENCE: Confirm/Decompose")
     (csetf *terms* (cons (list 'RESULT: result 'RESULT 'TYPE:
                             result-type 'INFERENCE:CONFIRM) *terms*))
     (sd-confirm))
  ((eql result-type '#$Insufficient)
     (format t " RESULT TYPE: #$Insufficient, INFERENCE: Select2-3")
     (csetf *terms* (cons (list 'RESULT: result 'RESULT 'TYPE:
                             result-type 'INFERENCE:SELECT2-3) *terms*))
     (sd-select2-3))
  ((cor (eql result-type '#$Normal) (eql result-type '#$Distinguishing))
     (format t " RESULT TYPE: ~S, INFERENCE: NewHypothesis" result-type)
     (csetf *terms* (cons (list 'RESULT: result 'RESULT 'TYPE:
                            result-type 'INFERENCE:New_Hypothesis) *terms*))
     (sd-new-hypothesis result-type))
  (t (format t " RESULT TYPE: UNKNOWN!, INFERENCE: Diagnosis interrupted"))
 ) ;end of 'pcond'
)

;;; ********************************************
;;; ******** SubL CODE FOR SD-CONFIRM ************
;;; ********************************************
```

```
;;; Function : SD-CONFIRM
;;; Arguments: None
;;; Result   : If the current 'hypothesis' is a 'PCSUbSystem', it
;;;            is decomposed by calling function 'sd-decompose';
;;;            otherwise, it is a PCComponent and the diagnosis
;;;            terminates reporting this 'PCComponent' as faulty.
;;; Remarks  :

(define sd-confirm ()
 (csetq ask-result
  (fi-ask '(#$and
            (#$hypothesis ?HYP)
            (#$isa ?HYP #$PCComponent)) *defaultMt*))
 (pcond
  (ask-result
   (format t "~%~% DIAGNOSIS ENDED. FAULTY COMPONENT: ~S"
      (get-ask-binding (first ask-result) 1))
   (csetf *terms* (cons (list 'DIAGNOSIS 'ENDED. 'FAULTY 'COMPONENT:
                          (get-ask-binding (first ask-result) 1)) *terms*))
   (reverse *terms*))
  (t (format t " Decomposing...")
     (safe-fi :assert '(#$plausibleInference #$Decompose) *defaultMt*)
     (sd-decompose)
     (sd-select2-3))
 )
)

;;; *********************************************
;;; ******** SubL CODE FOR SD-NEW-HYPOTHESIS *****
;;; *********************************************

;;; Function: SD-NEW-HYPOTHESIS
;;; Arguments: 1. A #$ResultType
;;; Results:
;;; Remarks: This function is called by 'sd-confirm' with two types of
;;;          results, #$Normal and #$Distinguishing. If the result is
;;;          #$Normal, the function just changes the current
;;;          'hypothesis' to the next one (if one exists) and performs
;;;          the appropriate 'Test' by calling 'sd-select2-3'. If the result is
;;;          #$Distinguishing, then it changes the current
;;;          'hypothesis' as before, but doesn't perform any 'Test', as
;;;          the last 'Test' performed indicates that the next
;;;          'hypothesis' is faulty. Therefore, it calls the
;;;          'sd-confirm' function.

(define sd-new-hypothesis (result-type)

;; get the next hypothesis from the 'testAfter' assertions, if anyone
;; is left
 (csetq ask-result
       (fi-ask '(#$and
                  (#$hypothesis ?HYP)
                  (#$testAfter ?HYP ?NEW)HYP)) *defaultMt*))
 (pcond
  ((null ask-result) (format t "~% ~%I'M SORRY. THERE IS NO ALTERNATIVE
                        SYSTEM TO BE CONSIDERED. DIAGNOSIS FAILED"))
  (t (csetq hypothesis (get-ask-binding (first ask-result) 1))
     (csetq new_hypothesis (get-ask-binding (first ask-result) 2))
     (fi-unassert (list '#$hypothesis hypothesis) *defaultMt*)
     (fi-unassert (list '#$testAfter hypothesis new_hypothesis) *defaultMt*)
     (safe-fi :assert (list '#$hypothesis new_hypothesis) *defaultMt*)
     (pcond
      ((eql result-type '#$Normal) (sd-select2-3))
      ((eql result-type '#$Distinguishing) (sd-confirm))
     )) ;end of inner 'pcond' and 't' clause
 ) ;end of 'pcond'
```

```
)

;;; *********************************************
;;; ******** SubL CODE FOR SD-SELECT1 ********
;;; *********************************************

;;; Function: SD-SELECT1
;;; Arguments: None.
;;; Results: Asserts the '(#$hypothesis #$PCSystem)' fact to start the
;;;          Systematic Diagnosis problem solving method
;;; Remarks:

(define sd-select1 ()
 (safe-fi :assert '(#$hypothesis #$PCSystem) *defaultMt*)
 (sd-select2-3) ; ask the user what the general complaint is
)

;;; *********************************************
;;; ******** SubL CODE FOR SD-SELECT2-3 *************
;;; *********************************************

;;; Function : SD-SELECT2-3
;;; Arguments: None
;;; Result   :
;;; Remarks  : According to the situation, there may be a lot of
;;; 'possibleTest' assertions in the KB, but only one of them is the
;;; one that must be performed next. This Test is distinguished by the
;;; fact  that it is the only one that doesn't have a corresponding
;;; 'resultOfTest' assertion as it is not yet carried out. The
;;; 'sd-select2-3' function must therefore find this Test and pass it and
;;; its possible results ('possibleresultOfTest' assertions) to the
;;; 'get-test-result' function.


(define sd-select2-3 ()

;;get the current 'hypothesis'
 (csetq hypothesis (fi-ask '(#$hypothesis ?HYP) *defaultMt*))
 (csetq hypothesis (get-ask-binding (first hypothesis) 1))
 (format t "~%~%HYPOTHESIS: ~S" hypothesis)
 (csetf *terms* (cons (list  'HYPOTHESIS: hypothesis) *terms*))

;; Get all possible tests
 (csetq possible-tests
   (fi-ask '(#$possibleTest ?TEST) *defaultMt*))

;; keep the one that doesn't have a corresponding 'resultOfTest' assertion
 (cdo
  ((test
     (get-ask-binding (first possible-tests) 1)
     (get-ask-binding (first possible-tests) 1))
   ) ; end of variables
  ((cnot (fi-ask (list '#$resultOfTest test '?R) *defaultMt*)) t) ;exit condition
  (csetq possible-tests (rest possible-tests))
 )
 (csetq test (get-ask-binding (first possible-tests) 1))
 (csetf *test* test)

;; Get the possible results for this test
 (csetq possible-results
   (fi-ask
    (list
     '#$possibleResultOfTest (list '#$TestFn (second test) (third test)
                                   (fourth test)) '?VAL '?TYPE) *defaultMt*))
 (get-test-result test possible-results)
```

```
)

;;; *********************************************
;;; ******** SubL CODE FOR GET-TEST-RESULT *****
;;; *********************************************

;;; Function : GET-TEST-RESULT
;;; Arguments: 1. A 'Test' structure, which is a list of the form:
;;;
;;; (TestFn PC_SUBSYSTEM TEST_ACTION POSSIBLE_OBSERVABLE)
;;;
;;;            2. A list of the form:
;;;
;;; (
;;;  ((?VAL . RESULT_1) (?TYPE . TYPE_1))
;;;      ...
;;;  ((?VAL . RESULT_n) (?TYPE . TYPE_n))
;;; )
;;;
;;; Result   : An 'resultOfTest' Assertion in the KB with the actual result of
;;;            the test.
;;; Remarks  : For the moment, it is the 'menu' function that performs
;;;            the actual task as there in no way to get input from
;;;            the user when in the SubL interactor.

(define get-test-result (test possible-results)
 (present-test-parameters test)
 (present-test-results possible-results)
 (format t "~%~%Please, type '(menu [number_of_result])'")
; (input-test-result no_of_choices)
 (reverse *terms*) ;return as RESULT a list of all CYC terms appearing on the screen

;;; For the moment, we don't know how to interact with the user when in the
;;; SubL Interactor interface. Therefore, the user must give the command
;;; '(menu <number_of_result>)' to interact with the SubL code.
)


;;; **********************************************
;;; ******** SubL CODE FOR PRESENT-TEST-PARAMETERS*
;;; **********************************************

;;; Function : PRESENT-TEST-PARAMETERS
;;; Arguments: A 'Test' structure, which is a list of the form:
;;;
;;; (TestFn PC_SUBSYSTEM TEST_ACTION POSSIBLE_OBSERVABLE)
;;;
;;; Result   : Prints to the screen the current 'test' parameters, i.e,
;;;            the PCSubSystem, TestAction and PossibleObservable.
;;; Remarks  :

(define present-test-parameters (test)
 (format t "~%NEW TEST: ~%PCSubSystem: ~S ~%TestAction: ~S
   ~%Observable: ~S ~%" (second test) (third test) (fourth test))
 (csetf *terms* (cons (cons 'NEW (cons 'TEST test)) *terms*))
 )

;;; *********************************************
;;; ******** SubL CODE FOR PRESENT-TEST-RESULTS **
;;; *********************************************

;;; Function : PRESENT-TEST-RESULTS
;;; Arguments: A list of the form:
;;;
;;; (
;;;  ((?VAL . RESULT_1) (?TYPE . TYPE_1))
;;;      ...
```

```
;;;  ((?VAL . RESULT_n) (?TYPE . TYPE_n))
;;; )
;;;
;;; Result   : An enumarated menu of all possible results of the current 'test'
;;; Remarks  :

(define present-test-results (possible-results)

;; Get possible results
 (csetq results (mapcar #'get-ask-binding
                        possible-results
                        (position-list (length possible-results) 1)))

;; Get possible results' types (Not needed for the moment. The result
;; type is retreived by the 'sd-compare' function).

; (csetq result_types (mapcar #'get-ask-binding
;                               possible-results
;                               (position-list (length possible-results) 2)))

 (csetq counter 1)
 (cdolist (result results 't)
   (format t "~A. ~S~%" counter result)
   (csetf *terms* (cons (cons counter result) *terms*))
   (csetq counter (+ counter 1))
 )
 (csetf *no_of_choices* (- counter 1)) ;needed by 'menu'
 (csetf *results* results) ;needed by 'menu'
; (csetf *result_type* result_types) ; needed by 'menu'
)

;;; *********************************************
;;; ******** SubL CODE FOR POSITION-LIST *
;;; *********************************************

;;; Function : POSITION-LIST
;;; Arguments: 1. A number, n, indicating the number of bindings (lists of
;;;              doted pairs), in an ask-result of an 'fi-ask' function.
;;;            2. A number, k (1 =< k =< n), indicating which BINDING must
;;;              be returned by the 'get-ask-binding' function.
;;; Result   : A list of n elements equal to k.
;;; Remarks  : An auxiliary function. Creates the second arcument to be used
;;;            in a 'mapping' function which collects a list of BINDINGS for
;;;            the same ask variable.

(define position-list (n k)
 (csetq res ())
 (cdotimes (c n res)
  (csetq res (cons k res))
 )
 res
)


;;; *********************************************
;;; ******** SubL CODE FOR GET-ASK-BINDING ***
;;; *********************************************

;;; Function : GET-ASK-BINDING
;;; Arguments: 1. A list of dotted pairs of the form:
;;;
;;; ((?VAR1 . BINDING1)
;;;  (?VAR2 . BINDING2)
;;;  ...
;;;  (?VARn . BINDINGn)
;;; )
```

```
;;;            2. A number defining which BINDING must be returned.

;;; Result   : POSSIBLE_OBSERVABLE_VALUE
;;; Remarks  :

(define get-ask-binding (bindings_list bind_no)
 (rest (nth (- bind_no 1) bindings_list))
)


;;; *********************************************
;;; ******** SubL CODE FOR MENU ****************
;;; *********************************************

;;; Function : MENU
;;; Arguments: 1. A number from the menu of the possible results (local)
;;;            2. The list of possible results (global variable *results*)
;;;            3. The number of choices (global variable *no_of_choices*)
;;; Result   : Asserts into the KB a 'resultOfTest' assertion
;;; Remarks  :

(define menu (selection)
 (pcond
  ((cor (< selection 1) (> selection *no_of_choices*))
    (format t "~%~%You must give as an argument, a number between 1-~A"
     *no_of_choices*))
  (t (csetq test (fi-ask '(#$possibleTest ?TEST) *defaultMt*))
     (csetq test (get-ask-binding (first test) 1))
;     (csetf *result_type* (nth (- selection 1) *result_type*))
     (safe-fi :assert
      (list '#$resultOfTest test  (nth (- selection 1) *results*)) *defaultMt*))
 )
 (systematic '#$PCSystem)
)




;;; The 'decompose' inference in KADS Systematic Diagnosis PSM takes as input
;;; the current #$PCSubSystem (#$hypothesis PC_SUBSYSTEM) and decomposes it
;;; into its functional subsystems (#$functionalPartOf PC_SUBSYSTEM PART),
;;; generating new hypotheses (#$possibleHypotheses PART).


;;; *********************************************
;;; ******** SubL CODE FOR  SD-DECOMPOSE *****
;;; *********************************************

;;; Function : SD-DECOMPOSE
;;; Arguments: None.
;;; Result   :
;;; Remarks  :

(define sd-decompose ()
 ;; Before un-asserting the current hypothesis, its functional parts
 ;;  must be saved
 (csetq ask-result (fi-ask '(#$possibleHypotheses ?H) *defaultMt*))
 (csetq in_hypotheses
  (mapcar #'get-ask-binding ask-result (position-list (length ask-result) 1)))

 ;; Before un-asserting the current hypothesis, the order of its subsystems
 ;; diagnosis must be saved
 (csetq first_to_test (fi-ask '(#$testFirst ?SYS) *defaultMt*))
 (csetq first_to_test (get-ask-binding (first first_to_test) 1))
 (csetq afters (fi-ask '(#$testAfter ?S1 ?S2) *defaultMt*))
 (csetq s1_list
```

```
        (mapcar #'get-ask-binding afters (position-list (length afters) 1)))
 (csetq s2_list
        (mapcar #'get-ask-binding afters (position-list (length afters) 2)))

;; Store hypothesis and un-assert it
 (csetq hypothesis (fi-ask '(#$hypothesis ?H) *defaultMt*))
 (csetq hypothesis (get-ask-binding (first hypothesis) 1))
 (fi-unassert (list '#$hypothesis hypothesis) *defaultMt*)

;; Assert possibleHypotheses.
 (cdolist (system in_hypotheses t)
  (safe-fi :assert (list '#$possibleHypotheses system) *defaultMt*)
 )

;; Assert diagnosis order
; (safe-fi :assert (list '#$testFirst first_to_test) *defaultMt*) ;unnecessary
 (cdolist (s1 s1_list t)
  (csetq s2 (first s2_list))
  (csetq s2_list (rest s2_list))
  (safe-fi :assert (list '#$testAfter s1 s2) *defaultMt*)
 )

;; Un-assert the (#$plausibleInference #$Decompose) assertion.
 (fi-unassert '(#$plausibleInference #$Decompose) *defaultMt*)

;; Assert the new hypothesis
 (safe-fi :assert (list '#$hypothesis first_to_test) *defaultMt*)
)

;;; ********************************************
;;; ******** SubL CODE FOR  SD-RESET ****************
;;; ********************************************

;;; Function : SD-RESET
;;; Arguments: None
;;; Result   : Resets all assertions regarding the following
;;; predicates:
;;;     '#$diagnosisContext
;;;     '#$hypothesis',
;;;     '#$possibleHypotheses'
;;;     '#$resultOfTest'
;;;     '#$testFirst'
;;;     '#$testAfter'
;;;     '#$diagnosisContext' (dependant from #$resultOfTest)
;;;     '#$possibleTest'     (dependant from #$hypothesis & #$resultOfTest)

;;; Remarks  :

(define sd-reset ()
 ;; Set the global variables
 (csetf *use-local-queue?* NIL)
 (csetf *defaultMt* '#$PCDiagnosisMt)
 (csetf *test* nil)
 (csetf *results* nil)
 (csetf *no_of_choices* 0)

 ;;UN-ASSERT #$diagnosisContext
 (csetq diagnosisContext (fi-ask '(#$diagnosisContext ?C) *defaultMt*))
 (csetq diagnosisContext (get-ask-binding (first diagnosisContext) 1))
 (fi-unassert (list '#$diagnosisContext diagnosisContext) *defaultMt*)

 ;; UN-ASSERT #$hypothesis
 (csetq hypothesis (fi-ask '(#$hypothesis ?H) *defaultMt*))
 (csetq hypothesis (get-ask-binding (first hypothesis) 1))
 (fi-unassert (list '#$hypothesis hypothesis) *defaultMt*)
```

```
 ;; UN-ASSERT #$possibleHypotheses
 (csetq ask-result (fi-ask '(#$possibleHypotheses ?H) *defaultMt*))
 (csetq results
  (mapcar #'get-ask-binding ask-result (position-list (length ask-result) 1)))
 (cdolist (result results t)
  (fi-unassert (list '#$possibleHypotheses result) *defaultMt*)
 )

 ;; UN-ASSERT #$resultOfTest
 (csetq ask-result (fi-ask '(#$resultOfTest ?T ?R) *defaultMt*))
 (cdolist (bindings ask-result t)
  (fi-unassert
    (list '#$resultOfTest (get-ask-binding bindings 1)
                          (get-ask-binding bindings 2)) *defaultMt*)
 )

 ;;UN-ASSERT #$testFirst
 (csetq first_to_test (fi-ask '(#$testFirst ?SYS) *defaultMt*))
 (csetq first_to_test (get-ask-binding (first first_to_test) 1))
 (fi-unassert (list '#$testFirst first_to_test) *defaultMt*)

 (csetq afters (fi-ask '(#$testAfter ?S1 ?S2) *defaultMt*))
 (csetq s1_list
       (mapcar #'get-ask-binding afters (position-list (length afters) 1)))
 (csetq s2_list
       (mapcar #'get-ask-binding afters (position-list (length afters) 2)))
 (cdo
  ((s1 (first s1_list) (first s1_list))
   (s2 (first s2_list) (first s2_list))
   (s1_list (rest s1_list) (rest s1_list))
   (s2_list (rest s2_list) (rest s2_list))
  ) ;end of variables
  ((null s1)) ;when no more couples of s1,s2
  (fi-unassert (list '#$testAfter s1 s2) *defaultMt*)
 )
)
```

Appendix D

The CYC KE-Text for PC/Automobile Domains

```
;;; PROJECT:  628-Implementing  Problem Solving Methods (PSMs)  in Cyc
;;; FILENAME: systematic_diagnosisKE.txt
;;; AUTHOR:   Dimitrios Sklavakis
;;; PURPOSE:  Contains Cyc's Knowledge Entering  (KE) text defining the
;;;           general knowledge   for the implementation  of  the
;;;           Systematic Diagnosis (Localisation) PSM   from  KADS
;;;           methodology for Personal Computers (PCs) and Automobiles.

;;; LAST UPDATED: 05/08/1998.

;**************** THE #$SystematicDiagnosisMt MICROTHEORY ***********

;;; The general knowledge for  performing Systematic Diagnosis will be
;;; grouped in the #$SystematicDiagnosisMt microtheory.  This one is a
;;; more   specific   microtheory   than the  #$BaseKB   microtheory,
;;; i.e.  (#$genlMt #$SystematicDiagnosisMt #$BaseKB) and more general
;;; than  the    #$PCDiagnosisMt       and     #$AutomobileDiagnosisMt
;;; microtheories:

;;; (#$genlMt   #$PCDiagnosisMt   #$SystematicDiagnosisMt)   (#$genlMt
;;; #$AutomobileDiagnosisMt #$SystematicDiagnosisMt)

constant: SystematicDiagnosisMt.
isa:  Microtheory.
genls:  BaseKB.
comment: "#$SystematicDiagnosisMt is  the #$Microtheory that  contains
all   the assertions  about   performing  KADS' Systematic  Diagnosis
(Localisation) problem solving method.".



; **************** SYSTEM MODEL KNOWLEDGE ****************

Default Mt: SystematicDiagnosisMt.

constant: SubSystem.
isa: Collection.
genls: CompositeTangibleAndIntangibleObject.
comment: "The collection of all Psub-systems, like the
#$VideoSystem, #$PowerSystem, #$KeyboardSystem. Each instance of
#$SubSystem may include several #$Components and/or other
#$SubSystems. Different #$SubSystems may include the same
#$Components. In the context of #$SystematicDiagnosisMt any #$SubSystem is
```

```
an intermediate level of analysis for the system model; the
diagnosis continues until a faulty #$Component is located".

constant: Component.
isa: Collection.
genls: SubSystem.
comment: "The collection of all components such as the
#$PowerSupply, #$VideoCard, #$FloppyDiskDrive. In the context of
#$SystematicDiagnosisMt any #$Component is the lowest level of analysis for
the system model; the diagnosis terminates when a faulty
#$Component is located.".

;;; For Systematic Diagnosis,  a predicate is needed to express the
;;; hierarchical system model, consisting of #$SubSystems and
;;; #$Components. However, this predicate is domain dependent and
;;; therefore will be defined seperately in each domain-specific
;;; theory. For example, in #$PCDiagnosisMt it is #$functionalPartOf,
;;; while in #$AutomobileDiagnosisMt is #$physicalDecompositions.

constant: testFirst.
isa: UnaryPredicate.
arg1Isa: SubSystem.
comment: "This predicate is used to declare which SubSystem from
these occuring after a #$Decompose inference type will be the
the first to consider for diagnosis, i.e., the #$hypothesis.".

constant: testAfter.
isa: BinaryPredicate.
arg1Isa: SubSystem.
arg2Isa: SubSystem.
comment: "This predicate is used to declare the order of considering
subsystems for diagnosis. For example, (#$testAfter SUBSYSTEM1
SUBSYSTEM2) means that that the #$SubSystem SUBSYSTEM2 will
be considered for diagnosis immmediately after SUBSYSTEM1.".

constant: hypothesis.
isa: UnaryPredicate.
arg1Isa: SubSystem.
comment: "The predicate is used to record in the KB which
#$SubSystem is currently being diagnosed. E.g.,
#$hypothesis(#$VideoSystem) means that it is the #$VideoSystem that is
currently being checked for possible faults.".

constant: possibleHypotheses.
isa: UnaryPredicate.
arg1Isa: SubSystem.
comment: "The predicate is used to record in the KB which
#$SubSystems are currently candidates for being diagnosed. E.g.,
#$possibleHypotheses(#$VideoCard) means that the #$VideoCard is
a candidate to be checked for possible faults.".

;**************** SYSTEM TESTING KNOWLEDGE ****************

;;; The basic tool in the Systematic Diagnosis problem-solving method
;;; (PSM), as well as in any other diagnostic PSM, for carrying out
;;; the diagnostic procedure, is various TESTS that must be done to
;;; provide information (knowledge) about the state of the
;;; system. This knowledge may concern the actual behaviour of the
;;; system's components, e.g. the absence of electric power, control
;;; information produced by the system, e.g., beep codes or screen
;;; messages. Conceptually, a TEST is a question that the user must
;;; make to the system under diagnosis to extract knowledge about
;;; it. Here, it is implemented as a structure consisting of three
;;; other concepts:
;;; 1. The #$SubSystem to which it is related, i.e., to which the
;;;    question is adressed.
```

```
;;; 2. The #$TestAction which is the action one has really to perform
;;;    for the test
;;; 3. The #$PossibleObservable (system variable) that the TEST is
;;;    asking about .

;;; Although not part of a TEST structure, there is a fourth concept
;;; related with it, the  #$PossibleObservableValue (system variable
;;; value) which is the result (answer) of the TEST's question.

;;; Each TEST is represented as a non-atomic term (NAT) in CycL with
;;; the use of a #$NonPredicateFunction, #$TestFn, which takes as
;;; arguments instances of the three constituent concepts and returns
;;; a TEST structure, Schematically:

;;; (#$TestFN #$SubSystem #$TestAction #$PossibleObservable) -> #$Test

Default Mt: SystematicDiagnosisMt.

constant: Test.
isa: Collection.
genls: InformationBearingThing.
comment: "The basic tool in the Systematic Diagnosis problem-solving
method (PSM), as well as in any other diagnostic PSM, for carrying out
the diagnostic procedure, is various TESTS that must be done to
provide information (knowledge) about the state of the system. This
knowledge may concern the actual behaviour of the system's components,
e.g. the absence of electric power, control information produced by
the system, e.g., beep codes or screen messages. Conceptually, a TEST
is a question that the user must make to the system under diagnosis to
extract knowledge about it. Here, it is implemented as a structure
consisting of three other concepts: 1. The #$SubSystem to which it
is related, i.e., to which the question is adressed, 2. The
#$TestAction which is the action one has really to perform for the
test and 3. The #$PossibleObservable (system variable) that the TEST
is asking about.".


constant: TestFn.
isa: NonPredicateFunction.
arity: 3.
arg1Isa: SubSystem.
arg2Isa: TestAction.
arg3Isa: PossibleObservable.
resultIsa: Test.
comment: "Every #$Test is a structure consisting of three
concepts. The #$SubSystem to which it is related, the actual
#$TestAction that must be performed and the #$PossibleObservable
(system variable) that is being observed.Each TEST is represented as a
non-atomic term (NAT) in CycL
with the use of the #$NonPredicateFunction, #$TestFn, which takes as
arguments instances of the three constituent concepts and returns a
TEST structure, Schematically:
(#$TestFN #$SubSystem #$TestAction #$PossibleObservable) -> #$Test".

constant: possibleTest.
isa: UnaryPredicate.
arg1Isa: Test.
comment: "The #$Tests available to be performed in any stage of the
Diagnosis.".

constant: TestAction.
isa: Collection.
genls: PurposefulAction.
comment: "The collection of all possible test actions that may be
performed from the user on a #$SubSystem to determine the
#$PossibleObservableValues of
```

```
a #$PossibleObservable. These values are compared to the expected
ones. If they are different, the fault lies somewhere in the
#$SubSystem which is further decomposed and its parts are
checked one by one. If not, the fault lies in another #$SubSystem.".

constant: PossibleObservable.
isa: Collection.
genls: AttributeType.
comment: "The colection of system variables (possible observables)
which are used to decide if the currently checked #$SubSystem actually
contains a faulty #$Component or not.".

constant: PossibleObservableValue.
isa: Collection.
genls: AttributeValue.
comment: "The collection of all possible values of all
#$PossibleObservables. In terms of the Systematic Diagnosis
problem-solving method, the instances of #$PossibleObservable
correspond to the system variables that one can test during diagnosis and the
instances of #$PossibleObservableValue correspond to the possible
outcomes of these tests.".

constant: ObservableValueFn.
isa: NonPredicateFunction.
arity: 1.
arg1Isa: Thing.
resultIsa: PossibleObservableValue.
comment: "Although the collection of all terms that are used as
possible answers to the #$Tests of the Systematic Diagnosis problem
solving method (PSM) are collected in the #$PossibleObservableValue
collection, in general anything can be an instance of
#$PossibleObservableValue. For example, the #$HardDiskDrive can be an
instance of this collection. It only depends on the questions one
makes. Anything can be an answer to a question. For this reason, the
#$ObservableValueFn function is introduced. It can take anything as an
argument and make it play the role of an instance of
#$PossibleObservableValue. This function makes the existence
of #$PossibleObservableValue seem redundant, however this latter
collection plays a significant role in the analysis of the Systematic
Diagnosis PSM and therefore has a reason to exist.".

constant: ResultType.
isa: Collection.
genls: AttributeValue.
comment: "The collection of all various types of
#$PossibleObservableValues. These types are characterised from the
kind of conclusions they lead relative to the #$SubSystem being
currently diagnosed ( #$hypothesis(#$SubSystem) ). E.g., such a type
can be #$Normal which denotes that the #$SubSystem currently being
diagnosed is not faulty and therefore must be discarded as a
hypothesis and a new hypothesis must be selected.".

constant: possibleResultOfTest.
isa: Predicate.
arity: 3.
arg1Isa: Test.
arg2Isa: PossibleObservableValue.
arg3Isa: ResultType.
comment: "The predicate is correlating an  individual #$Test with its
actual result and the type of this result. The assertion
possibleResultOfTest(TEST VALUE TYPE)
express the fact that for the specific TEST, VALUE is a possible
result of type TYPE. E.g., possibleResultOfTest((TestFn
PowerSystem ConfirmSensorially ElectricPower) Yes Normal) indicates that
it is #$Normal to observe the existence of #$ElectricPower when
diagnosing the #$PowerSystem. Of course, this immediately would imply
```

```
that the #$PowerSystem is not faulty and therefore should be discarded
as a #$hypothesis.".

constant: resultOfTest.
isa: BinaryPredicate.
arg1Isa: Test.
arg2Isa: PossibleObservableValue.
comment: "The predicate is correlating an  individual #$Test with its
actual result. The assertion resultOfTest(?TEST ?VALUE) means that
?VALUE is the actual result of ?TEST.".

;******** DEFINITIONS OF INSTANCES FOR  #$TestAction COLLECTION ********

constant: ConfirmSensorially.
isa: TestAction.
comment: "The action of confirming the existence of a
#$PossibleObservable only by one's senses, e.g., visually,
acoustically.".


constant: CheckIndependently.
isa: TestAction.
comment: "This #$TestAction means that the user performing
diagnosis must check the function of the related #$SubSystem
isolated from the rest of the #$SubSystem of which it is a
part of. The way to do that is not specifically described
by the name of the action. It is assumed that the user has some
knowledge for performing such isolated tests. E.g., to test the
#$PowerSocket one can plug another device - known to be working -  in
it and confirm that the device has #$ElectricPower.".

constant: Remove.
isa: TestAction.
comment: "This #$TestAction means that the user must remove the
related #$SubSystem from the #$SubSystem of which it is a
part of".

constant: Replace.
isa: TestAction.
comment: "This #$TestAction means that the user must replace the
related #$SubSystem with a new one".

constant: TroubleshootComponent.
isa: TestAction.
comment: "This #$TestAction means that the diagnosis reached at the
level of a specific PCComponent but there is not sufficient
information to confirm that it is faulty. Therefore, the user must
enter the stage of troubleshooting it specifically.".

;***** DEFINITIONS OF INSTANCES FOR $PossibleObservable COLLECTION ****

constant: ProblemContext.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the general context of
diagnosis, i.e., #$BootTime, #$RunTime, #$ComponentSpecific. The
value of this #$PossibleObservable determines which rules are
applicable, appearing as a condition in their antecedent part".


;***** DEFINITIONS OF INSTANCES FOR $PossibleObservableValue COLLECTION ****

constant: RunTime.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed occured during run-time, i.e., during its normal operation".
```

```
constant: Yes.
isa: PossibleObservableValue.
comment: "Most of the #$Tests have as a possible result only 'Yes' or 'No'.".

constant: No.
isa: PossibleObservableValue.
comment: "Most of the #$Tests have as a possible result only 'Yes' or 'No'.".

constant: None.
isa: PossibleObservableValue.
comment: "This kind of result indicates that none of the alternative
results of a specific #$Test is observed.".

;***** DEFINITIONS OF INSTANCES FOR $ResultType COLLECTION ****

constant: Normal.
isa: ResultType.
comment: "This type of result indicates that the result is normally
expected when the #$SubSystem related with it is working
properly. Such a kind of result implies that the #$SubSystem must be
discarded as a #$hypothesis.".

constant: NotNormal.
isa: ResultType.
comment: "This type of result indicates that the result is not normally
expected when the #$SubSystem related with it is working
properly. Such a kind of result implies that the fault lies in the
#$SubSystem which is the current #$hypothesis.".

constant: Insufficient.
isa: ResultType.
comment: "This type of result indicates that the result cannot
undoubtedly indicate either the normal function or the malfunction of
the #$SubSystem related with it. Such a kind of result implies that
further testing is necessary to decide about the functional status of
the #$SubSystem which is the current #$hypothesis.".

constant: Distinguishing.
isa: ResultType.
comment: "This type of result occurs in a situation where there are
two components that are probably faulty and the only way to find
which, is to test one of them. In this case, a result of type
#$Distinguishing indicates simultaneously two things. First, that the
#$SubSystem hypothesised as faulty is not such and second, that the faulty
one is the other alternative #$SubSystem.".


;;; ********* IMPLEMENTATION OF KADS SYSTEMATIC DIAGNOSIS PSM ********

;;; The Task Structure for Systematic Diagnosis (pseudo-code) is:

;;; Systematic Diagnosis(+complaint,+possible observables,-hypothesis) by
;;;  select1(+complaint, -system model)
;;;  REPEAT
;;;   decompose(+system model, -hypothesis)
;;;   WHILE number of hypotheses > 1
;;;    select2(+possible observables, -variable value)
;;;    select3(+hypothesis, -norm)
;;;    compare(+variable value, +norm, -difference)
;;;   system model <- current decomposition level of system model
;;;  UNTIL confirm(+hypothesis), i.e. system model cannot be decomposed further

;;; The user interaction in CYC will be done from the SubL Interactor
;;; interface, as it is not possible to get any input/output
;;; interaction between the user and the SubL code from the ASK
```

```
;;; interface. The whole Task Structure will be implemented as a SubL
;;; function, 'systematic', which will be responsible for calling the
;;; appropriate SubL functions that will implement the corresponding
;;; inferences. In fact, the 'select1', 'select2', and 'select3'
;;; inferences will be implemented as FORWARD rules in the
;;; CYC KB. "Forward" means that, according to the results of 'Tests'
;;; that the user is asked to give, these rules automatically assert
;;; new facts in the KB. These facts describe which are the next
;;; #$PossibleObservables and Variables that must be tested ('select2'
;;; inference), what should be done according to the result ('select3'
;;; and 'compare' inferences), e.g., if another test for the same
;;; hypothesis should be performed or if the current hypothesis should
;;; be rejected or the current hypothesis must be decomposed further
;;; or if the faulty component was found ('confirm' inference).


;;; The following three (3) constant definitions introduce the
;;; #$plausibleInference predicate and #$Decompose inference type of
;;; KADS. These two are used in the antecendent part of the
;;; "decomposition" rules. They do not constitute control knowledge
;;; but domain role knowledge. In terms of implementation, they cause
;;; the forward "decomposition" rules to fire only when a Decompose
;;; inference has to be made.

Default Mt: SystematicDiagnosisMt.

constant: InferenceType-KADS.
isa: Collection.
genls:  PropositionalInformationThing.
comment: "The collection of all Inference Types of KADS methodology,
e.g., 'select', 'decompose', 'confirm'.".

constant: Decompose.
isa: InferenceType-KADS.
comment: "The #$Decompose inference type of KADS takes a structured
hierarchy of objects and gives a less or completely unstructured
collection of these objects. In its simplest form it is used for
breaking down existing knowledge structures, like hierarchies, where
there is no loss of objects but only the structure is removed.".

constant: plausibleInference.
isa: UnaryPredicate.
arg1Isa: InferenceType-KADS.
comment: "The #$plausibleInference predicate is used to record in the
KB which inference(s) can be next performed during the 'execution' of
a problem solving method.".

constant: diagnosisContext.
isa: UnaryPredicate.
arg1Isa: PossibleObservableValue.
comment: "This predicate records the current problem-solving context
in a specific diagnosis domain, e.g., #$BootTime, #$StaringTime,
#$RunTime etc. Most of the rules in each domain are
context-dependent. This is reflected in the appearence of the
#$diagnosisContext predicate in their antecedent part.".

*********************************************************************
*********************************************************************
                END OF systematic_diagnosisKE.txt
*********************************************************************
*********************************************************************

;;; PROJECT:  628-Implementing  Problem Solving Methods (PSMs)  in Cyc
;;; FILENAME: automobile_diagnosisKE.txt
;;; AUTHOR:   Dimitrios Sklavakis
;;; PURPOSE:  Contains Cyc's Knowledge Entering  (KE) text defining the
```

```
;;;           Automobile domain specific knowledge   for the
;;;           implementation  of the Systematic Diagnosis
;;;           (Localisation) PSM   from  KADS methodology.

;*************** THE #$AutomobileDiagnosisMt MICROTHEORY ***************

;;; The  domain specific knowledge for performing   Automobile fault
;;; diagnosis  will be entered in the #$AutomobileDiagnosisMt
;;; microtheory, which is a more specific microtheory of the
;;; #$SystematicDiagnosisMt microtheory:

;;;  (#$genlMt #$AutomobileDiagnosisMt #$SystematicDiagnosisMt).

constant: AutomobileDiagnosisMt.
isa: Microtheory.
genlMt: SystematicDiagnosisMt.
comment: "#$AutomobileDiagnosisMt is the #$Microtheory that contains
all the assertions about performing Automobile fault diagnosis.".

;;; ************************************************
;;; * Domain specific Automobile diagnosis knowledge *
;;; ************************************************

Default Mt: AutomobileDiagnosisMt.

; *************** THE SYSTEM MODEL ***************

constant: AutomobileSubSystem.
isa: Collection.
genls: SubSystem.
comment: "The collection of all  Automobile sub-systems, like the
#$IgnitionSystem. Each instance of #$AutomobileSubSystem may include
several #$AutomobileComponents and/or other
#$AutomobileSubSystems. Different #$AutomobileSubSystems may include
the same #$AutomobileComponents. In the context of
#$AutomobileDiagnosisMt any #$AutomobileSubSystem is an intermediate
level of analysis for the #$Automobile; the diagnosis continues until
a faulty #$AutomobileComponent is located".


constant: AutomobileComponent.
isa: Collection.
genls: AutomobileSubSystem Component.
comment: "The collection of all Automobile components such as the
#$SparkPlugs, #$Points, #$RotorArm. In the context of
#$AutomobileDiagnosisMt any #$AutomobileComponent is the lowest level
of analysis for the #$Automobile; the diagnosis terminates when a
faulty #$AutomobileComponent is located.".

constant: AutomobileSystem.
isa: Individual AutomobileSubSystem.
comment: "The #$AutomobileSystem is used to refer to the #$Automobile as
an #$AutomobileSubSystem. It includes the following #$AutomobileSubSystems:
#$IgnitionSystem etc.".

Constant: IgnitionSystem.
isa: AutomobileSubSystem.

Constant: HighTensionCircuit.
isa: AutomobileSubSystem.

Constant: LowTensionCircuit.
isa: AutomobileSubSystem.

Constant: Distributor.
isa: AutomobileSubSystem.
```

```
Constant: SparkPlugs.
isa: AutomobileComponent.

Constant: HighTensionWiring.
isa: AutomobileComponent.

Constant: Points.
isa: AutomobileComponent.

Constant: RotorArm.
isa: AutomobileComponent.

Constant: VacuumAdvance.
isa: AutomobileComponent.

Constant: CentrifugalWeights.
isa: AutomobileComponent.

;**************** SYSTEM TESTING KNOWLEDGE ****************

;****** DEFINITIONS OF INSTANCES FOR  #$PossibleObservable COLLECTION ********

Constant: FuelConsumption. ;;PossibleObservableValue: Integer
isa: PossibleObservable.   ;;Normal value: 7.0 (miles/litre)

;;HighTensionCircuit test
Constant: EngineMisfire.  ;;PossibleObservableValue: Yes/No
isa: PossibleObservable.  ;;Normal value: No

;;LowTensionCircuit test
Constant: EngineStarts.  ;;PossibleObservableValue: Yes/No
isa: PossibleObservable. ;;Normal value: Yes

;;Distributor test
Constant: Acceleration0to60.  ;;PossibleObservableValue: Integer
isa: PossibleObservable.      ;;Normal value: 15 (SecondsDuration)

;;SparkPlugs test
Constant: ColourOfCeramic.  ;;PossibleObservableValue:WhiteColor/
                            ;;GreyColor/BlackColor
isa: PossibleObservable.    ;;Normal value: WhiteColor

;;HighTensionWiring test
Constant: WiringSecurity.  ;;PossibleObservableValue: Secure/Insecure
isa: PossibleObservable.   ;;Normal value: Secure

;;Points test
Constant: SurfaceOfComponent. ;;PossibleObservableValue: Shiny/Dull/Rusty
isa: PossibleObservable.      ;;Normal value: Shiny

;;CentrifugalWeights test
Constant: StrobeTestResult. ;;PossibleObservableValue: Pass/Fail
isa: PossibleObservable.    ;;Normal value: Pass

;;VacuumAdvance test
Constant: VacuumTestResult. ;;PossibleObservableValue: Pass/Fail
isa: PossibleObservable.    ;;Normal value: Pass


Constant: WhiteColor.
isa: Color.
Constant: GrayColor.
isa: Color.
Constant: BlackColor.
isa: Color.
```

```
F: (genls Color PossibleObservableValue).


;***** DEFINITIONS OF INSTANCES FOR $PossibleObservableValue COLLECTION ****

constant: StartingTime.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed occured during the engine starting time, i.e., from the time
the driver turns the key on until the automobile starts moving".

Constant: Pass.
isa: PossibleObservableValue.

Constant: Fail.
isa: PossibleObservableValue.

Constant: Shiny.
isa: PossibleObservableValue.

Constant: Dull.
isa: PossibleObservableValue.

Constant: Rusty.
isa: PossibleObservableValue.

Constant: Secure.
isa: PossibleObservableValue.

Constant: Insecure.
isa: PossibleObservableValue.

;;; ***********************************
;;; ** Rules and Facts for Decompose **
;;; ***********************************

;; Every PART which is a physical decomposition of the hypothesis, HYP, is a
;; possible hypothesis.

direction: forward.
F: (implies
    (and
     (plausibleInference Decompose)
     (hypothesis ?HYP)
     (physicalDecompositions ?HYP ?PART))
    (possibleHypotheses ?PART)).

F: (physicalDecompositions AutomobileSystem IgnitionSystem).

F: (physicalDecompositions IgnitionSystem HighTensionCircuit).
F: (physicalDecompositions IgnitionSystem LowTensionCircuit).
F: (physicalDecompositions IgnitionSystem Distributor).

F: (physicalDecompositions HighTensionCircuit SparkPlugs).
F: (physicalDecompositions HighTensionCircuit HighTensionWiring).

F: (physicalDecompositions Distributor Points).
F: (physicalDecompositions Distributor RotorArm).
F: (physicalDecompositions Distributor VacuumAdvance).
F: (physicalDecompositions Distributor CentrifugalWeights).

;;; The following assertion permits for whole CYC formulae to appear
;;; as PossibleObservableValues
```

```
F: (genls CycFormula PossibleObservableValue).

;;; Rule to assert a (#$diagnosisContext ...) assertion. This assertion
;;; is introduced as a "shorthand" for the assertion:
;;;
;;; (resultOfTest
;;;    (TestFn AutomobileSystem ConfirmSensorially ProblemContext) ?PROBLEM)
;;;
;;; It is used as a premise in every rule which is applicable to the
;;; corresponding diagnosis context, i.e., #$StartingTime, #$RunTime etc.

Direction: forward.
F: (implies
    (resultOfTest
        (TestFn AutomobileSystem ConfirmSensorially ProblemContext) ?PROBLEM)
    (diagnosisContext ?PROBLEM)).



;;; *********************************************
;;; * Testing knowledge for the AutomobileSystem *
;;; *********************************************

Direction: forward.

F: (implies
    (hypothesis AutomobileSystem)
    (and
     (possibleTest (TestFn AutomobileSystem ConfirmSensorially ProblemContext))
     (possibleResultOfTest
      (TestFn AutomobileSystem ConfirmSensorially ProblemContext)
                                                   StartingTime NotNormal)
     (possibleResultOfTest
      (TestFn AutomobileSystem ConfirmSensorially ProblemContext)
                                                     RunTime NotNormal))).
;;; ***************************************************
;;; * Decomposition knowledge for the AutomobileSystem *
;;; ***************************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis AutomobileSystem)
     (plausibleInference Decompose))
    (and
     (testFirst IgnitionSystem))).

;;; *******************************************
;;; * Testing knowledge for the IgnitionSystem *
;;; *******************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis IgnitionSystem))
    (and
     (possibleTest (TestFn IgnitionSystem CheckIndependently FuelConsumption))
     (possibleResultOfTest
       (TestFn IgnitionSystem CheckIndependently FuelConsumption)
         (and (greaterThanOrEqualTo ?X 5) (greaterThanOrEqualTo 8 ?X)) Normal)
     (possibleResultOfTest
       (TestFn IgnitionSystem CheckIndependently FuelConsumption)
```

```
                                                (greaterThan ?X 8) NotNormal)
     (possibleResultOfTest
       (TestFn IgnitionSystem CheckIndependently FuelConsumption)
                                             (greaterThan 5 ?X) NotNormal))).

;;; *************************************************
;;; * Decomposition knowledge for the IgnitionSystem *
;;; *************************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis IgnitionSystem)
     (plausibleInference Decompose))
    (and
     (testFirst HighTensionCircuit)
     (testAfter HighTensionCircuit LowTensionCircuit)
     (testAfter LowTensionCircuit Distributor))).

;;; ***********************************************
;;; * Testing knowledge for the HighTensionCircuit *
;;; ***********************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis HighTensionCircuit))
    (and
     (possibleTest
      (TestFn HighTensionCircuit ConfirmSensorially EngineMisfire))
     (possibleResultOfTest
      (TestFn HighTensionCircuit ConfirmSensorially EngineMisfire) No Normal)
     (possibleResultOfTest
      (TestFn HighTensionCircuit ConfirmSensorially EngineMisfire)
                                                          Yes NotNormal))).

;;; *****************************************************
;;; * Decomposition knowledge for the HighTensionCircuit *
;;; *****************************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis HighTensionCircuit)
     (plausibleInference Decompose))
    (and
     (testFirst SparkPlugs)
     (testAfter HighTensionWiring))).

;;; ***********************************************
;;; * Testing knowledge for the LowTensionCircuit *
;;; ***********************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis LowTensionCircuit))
    (and
```

```
     (possibleTest (TestFn LowTensionCircuit ConfirmSensorially EngineStarts))
     (possibleResultOfTest
      (TestFn LowTensionCircuit ConfirmSensorially EngineStarts) Yes  Normal)
     (possibleResultOfTest
      (TestFn LowTensionCircuit ConfirmSensorially EngineStarts)
                                                               No NotNormal))).

;;; ****************************************
;;; * Testing knowledge for the Distributor *
;;; ****************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis Distributor))
    (and
     (possibleTest (TestFn Distributor CheckIndependently Acceleration0to60))
     (possibleResultOfTest
      (TestFn Distributor CheckIndependently Acceleration0to60)
                                                (greaterThan 15 ?X) Normal)
     (possibleResultOfTest
      (TestFn Distributor CheckIndependently Acceleration0to60)
                                  (greaterThanOrEqualTo ?X 15) NotNormal))).

;;; **********************************************
;;; * Decomposition knowledge for the Distributor *
;;; **********************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis Distributor)
     (plausibleInference Decompose))
    (and
     (testFirst Points)
     (testAfter Points VacuumAdvance)
     (testAfter VacuumAdvance CentrifugalWeights)
     (testAfter CentrifugalWeights RotorArm))).


;;; ***************************************
;;; * Testing knowledge for the SparkPlugs *
;;; ***************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis SparkPlugs))
    (and
     (possibleTest (TestFn SparkPlugs ConfirmSensorially ColourOfCeramic))
     (possibleResultOfTest
      (TestFn SparkPlugs ConfirmSensorially ColourOfCeramic) WhiteColor Normal)
     (possibleResultOfTest
      (TestFn SparkPlugs ConfirmSensorially ColourOfCeramic)
                                                        GrayColor NotNormal)
     (possibleResultOfTest
      (TestFn SparkPlugs ConfirmSensorially ColourOfCeramic)
                                                    BlackColor NotNormal))).

;;; **********************************************
```

```
;;; * Testing knowledge for the HighTensionWiring *
;;; ***********************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis HighTensionWiring))
    (and
     (possibleTest
      (TestFn HighTensionWiring ConfirmSensorially WiringSecurity))
     (possibleResultOfTest
      (TestFn HighTensionWiring ConfirmSensorially WiringSecurity)
                                                                Secure Normal)
     (possibleResultOfTest
      (TestFn HighTensionWiring ConfirmSensorially WiringSecurity)
                                                         Insecure NotNormal))).
;;; ************************************
;;; * Testing knowledge for the Points *
;;; ************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis Points))
    (and
     (possibleTest (TestFn Points ConfirmSensorially SurfaceOfComponent))
     (possibleResultOfTest
      (TestFn Points ConfirmSensorially SurfaceOfComponent) Shiny Normal)
     (possibleResultOfTest
      (TestFn Points ConfirmSensorially SurfaceOfComponent) Dull NotNormal)
     (possibleResultOfTest
      (TestFn Points ConfirmSensorially SurfaceOfComponent) Rusty NotNormal))).

;;; ************************************************
;;; * Testing knowledge for the CentrifugalWeights *
;;; ************************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
     (hypothesis CentrifugalWeights))
    (and
     (possibleTest
      (TestFn CentrifugalWeights ConfirmSensorially StrobeTestResult))
     (possibleResultOfTest
      (TestFn CentrifugalWeights ConfirmSensorially StrobeTestResult)
                                                                  Pass Normal)
     (possibleResultOfTest
      (TestFn CentrifugalWeights ConfirmSensorially StrobeTestResult)
                                                             Fail NotNormal))).

;;; *******************************************
;;; * Testing knowledge for the VacuumAdvance *
;;; *******************************************

Direction: forward.

F: (implies
    (and
     (diagnosisContext RunTime)
```

```
     (hypothesis VacuumAdvance))
    (and
     (possibleTest (TestFn VacuumAdvance ConfirmSensorially VacuumTestResult))
     (possibleResultOfTest
      (TestFn VacuumAdvance ConfirmSensorially VacuumTestResult) Pass Normal)
     (possibleResultOfTest
      (TestFn VacuumAdvance ConfirmSensorially VacuumTestResult)
                                                           Fail NotNormal))).

**********************************************************************
**********************************************************************
                END OF automobile_diagnosisKE.txt
**********************************************************************
**********************************************************************

;;; PROJECT:  628-Implementing  Problem Solving Methods (PSMs)  in Cyc
;;; FILENAME: pc_diagnosisKE.txt
;;; AUTHOR:   Dimitrios Sklavakis
;;; PURPOSE:  Contains Cyc's Knowledge Entering  (KE) text defining the
;;;           PC domain specific knowledge   for the implementation  of
;;;           the Systematic Diagnosis (Localisation) PSM   from  KADS
;;;           methodology.

;*************** THE #$PCDiagnosisMt MICROTHEORY ***************

;;; The  whole knowledge for performing   PC fault diagnosis  will be
;;; entered in the #$PCDiagnosisMt microtheory, which is a more specific
;;; microtheory of the #$SystematicDiagnosisMt microtheory:

;;;  (#$genlMt #$PCDiagnosisMt #$SystematicDiagnosisMt).

constant: PCDiagnosisMt.
isa: Microtheory.
genlMt: SystematicDiagnosisMt.
comment: "#$PCDiagnosisMt is the #$Microtheory that contains all the assertions
about performing Personal Computer(PC) fault diagnosis.".



;;; *****************************************
;;; * Domain specific PC diagnosis knowledge *
;;; *****************************************

Default Mt: PCDiagnosisMt.

; *************** THE SYSTEM MODEL ***************


constant: PCSubSystem.
isa: Collection.
genls: SubSystem.
comment: "The collection of all PC sub-systems, like the
#$VideoSystem, #$PowerSystem, #$KeyboardSystem. Each instance of
#$PCSubSystem may include several #$PCComponents and/or other
#$PCSubSystems. Different #$PCSubSystems may include the same
#$PCComponents. In the context of #$PCDiagnosisMt any #$PCSubSystem is
an intermediate level of analysis for the #$PersonalComputer; the
diagnosis continues until a faulty #$PCComponent is located".

constant: PCComponent.
isa: Collection.
genls: PCSubSystem Component.
comment: "The collection of all PC components such as the
#$PowerSupply, #$VideoCard, #$FloppyDiskDrive. In the context of
#$PCDiagnosisMt any #$PCComponent is the lowest level of analysis for
the #$PersonalComputer; the diagnosis terminates when a faulty
```

```
#$PCComponent is located.".

constant: PCSystem.
isa: Individual PCSubSystem.
comment: "The #$PCSystem is used to refer to the #$PersonalComputer as
a #$PCSubSystem. It includes the following #$PCComponents:
#$PowerSystem, #$VideoSystem, e.t.c.".

constant: PowerSystem.
isa: Individual PCSubSystem.
comment: "The power #$PCSubSystem. Includes the following
#$PCComponents: #$PowerSocket, #$PowerCable, #$PowerProtectionDevice
(optionally) and  #$PowerSupply.".

constant: PowerSocket.
isa: Individual PCComponent.
comment: "#$PowerSocket is the socket that provides electric power to
the PC. It is a component of the #$PowerSystem.".

constant: PowerCable.
isa: Individual PCComponent.
comment: "#$PowerCable is the cable that connects the #$PowerSocket
with the #$PowerSupply. It is a component of the #$PowerSystem.".

constant: PowerProtectionDevice.
isa: Individual PCComponent.
comment: "#$PowerProtectionDevice is any device (supproccessor, UPS)
connected between the #$PowerSocket and the #$PowerSupply to protect
the #$PersonalComputer from power failures. It is an optional
component of the #$PowerSystem.".

constant: PowerSupply.
isa: Individual PCComponent.
comment: "#$PowerSupply is the component located inside the
#$PersonalComputer case that supplies the #$MotherBoard with electriv
power. It is a component of the #$PowerSystem.".

constant: VideoSystem.
isa: Individual PCSubSystem.
comment: "The video #$PCSubSystem. Includes the following
#$PCComponents: #$MotherBoard, #$VideoCard, #$Monitor (and the
#$Speaker).".

constant: VideoCard.
isa: Individual PCComponent.
comment: "The #$VideoCard trasforms the video information to video
signal and sends it to the #$Monitor. It is a component of the
#$VideoSystem.".

constant: Monitor.
isa: Individual PCComponent.
comment: "The #$Monitor trasforms the video signal sent by the
#$VideoCard into visual image. It is a component of the
#$VideoSystem.".

constant: MotherBoard.
isa: Individual PCComponent.
comment: "The #$MotherBoard is the main #$PCComponent. Most of the
rest #$PCComponents are connected onto the #$MotherBoard and
controlled by it. It is actually a sub-system by itself as it includes
other components but in the context of PC diagnosis it will be
regarded as a #$PCComponent to avoid increasing the complexity of the
#$PersonalComputer analysis.".

constant: Speaker.
isa: Individual PCComponent.
```

```
comment: "The PC speaker. It refers to the
internal speaker that is connected on the motherboard and not the
external ones that are part of a multi-media system and require a
sound-card on which they are connected.".

constant: BIOSStartupSystem.
isa: Individual PCSubSystem.
comment: "The BIOS startup #$PCSubSystem. Includes the following
#$PCComponents: #$MotherBoard, #$VideoCard.".

constant: BIOSsettings.
isa: Individual PCComponent.
comment: "The Basic Input Output System (BIOS) settings record the
system parameters for all its operations. Wrong BIOS settings can be
responsible for a PC malfunction, e.g., the #$FloppyDiskDrive being
disabled and therefore being non-existant to the system.".

constant: MemorySystem.
isa: Individual PCSubSystem.
comment: "The memory #$PCSubSystem. Includes the following
#$PCComponents: #$RAM, #$MotherBoard.".

constant: RAM.
isa: Individual PCComponent.
comment: "The Random Access Memmory #$PCComponent. Usually, it is a
set of Single In-Line Memory Modules (SIMMs), plugged in special
positions on the #$MotherBoard.".

constant: FloppySystem.
isa: Individual PCSubSystem.
comment: "The #$FloppySystem consists of the #$FloppyDiskDrive and the
#$BIOSsettings for the enabling/disabling of the #$FloppyDiskDrive.".

constant: FloppyDiskDrive.
isa: Individual PCComponent.
comment: "The PC storage device which drives a removable floppy disk to
store/retrieve information.".

constant: HardDiskDrive.
isa: Individual PCComponent.
comment: "The PC main mass storage device. Usually it is fixed and
non-removable.".

constant: CDROMdrive.
isa: Individual PCComponent.
comment: "".

constant: PlugAndPlaySystem.
isa: Individual PCSubSystem.
comment: "This system comprises all peripherals that are connected to
the PC via expansion cards and they are (usually) automatically
recognised by MS Windows without any extra software drivers or
configuration procedures. However, sometimes there may be some
problems with their recognition. This #$PCSubSystem may has as
functional parts a wide variety of peripherals. In the current
implementation of Systematic diagnosis, it is regarded as consisting
of peripherals and their #$ExpansionCards. There isn't any further
decomposition into the specific peripherals as these are vary in each
configuration.".

constant: ExpansionCard.
isa: Individual PCComponent.
comment: "This PC component is used in conjunction with various
peripherals. In the current implementation, it is regarded as a
specific #$PCComponent, although in a specific PC configuration there
could be none, one or more expansion cards. Please, refer to the
```

```
#$PlugAndPlaySystem collection for more information.".

constant: PlugAndPlaySystem.
isa: Individual PCSubSystem.
comment: "It is the system responsible for loading the operating
system. In general, it comprises the #$FloppyDiskDrive with a floppy
disk containing the operating system (#$OSfloppyDisk) and the
#$HardDiskDrive, although a PC can be configured via the
#$BIOSsettings to use only one of them or both.".

constant: OSfloppyDisk.
isa: Individual PCComponent.
comment: "It is the floppy disk containing the operating system. It is
used by the #$BootSystem to load the OS from the #$FloppyDiskDrive.".

constant: BootSystem.
isa: Individual PCSubSystem.
comment: "It is the system responsible for loading the operating
system (OS). It includes the #$FloppyDiskDrive together with the floppy
disk containing the OS (#$OSfloppyDisk) and the #$HardDiskDrive. It
also includes the #$BootSequence-BIOSsetting which defines the sequence
in which these media will be used by the BIOS to load the OS.".

constant: functionalPartOf.
isa : TransitiveBinaryPredicate.
arg1Isa: PCSubSystem.
arg2Isa: PCSubSystem.
comment : "Predicate functionalPartOf is used to define a functional
          model of the PC under diagnosis
          functionalPartOf(WholeSubSystem PartialSubSystem) means
          that the PCSubSystem WholeSubSystem is using the function of
          PartialSubsystem to perforn its own function. E.g.,
          functionalPartOf(PowerSystem PowerSupply) means that for the
          PowerSystem to function the PowerSupply must function. This
          predicate is used to systematically disassemble the PC system into
          simpler PCSubSystems until a PCComponent is reached that it is
          faulty.".

;******** DEFINITIONS OF INSTANCES FOR  #$TestAction COLLECTION ********

constant: ChangeVoltage.
isa: TestAction.
comment: "This #$TestAction means that the user must change the
voltage setting in the #$PowerSupply. It is a specific #$TestAction
related only to the #$PowerSupply".

;****** DEFINITIONS OF INSTANCES FOR  #$PossibleObservable COLLECTION ********

constant: ElectricPower.
isa: PossibleObservable.
comment: "The electric power that any #$PCSystem needs to operate.".

constant: VoltageCorrect.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the voltage setting in
the #$PowerSupply being correct, i.e., 110V or 220V.".

constant: VideoSignal.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the existence of any
video signal on the #$Monitor screen.".

constant: SpeakerBeep.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to any beep pattern coming
out of the #$Speaker.".
```

```
constant: VideoBIOSMessage.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the display of the
video BIOS message.".

constant: BootContinues.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the booting process
cointinuing normally.".

constant: StartupScreen.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the display of the BIOS
stratup screen.".

constant: MemoryTest.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the memeory test
performed by the BIOS during boot-time.".

constant: ErrorMessage.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the display of an error
message on the screen.".

constant: ComponentProblem.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the occasion where a
specific #$PCComponent has reached which is possibly faulty and the
only way to decide about this involves elaborate and complex #$Tests,
which the current implementation of the Systematic diagnosis problem
solving method does not cover. Therefore, the user has to perform
these #$Tests either based on his knowledge or have a human expert
perform them.".

constant: AutoDetection-BIOSsetting.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the #$HardDiskDrive
auto-detection setting in the PC BIOS. It may be set to #$Auto for
automatic detection or to #$Manual, usually the first one.".

constant: BootSequence-BIOSsetting.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the BIOS setting which
controls the sequence of the media used to load the operating
system. It may be A:-C: for using first the #$FloppyDiskDrive and then
the #$HardDiskDrive or C:-A: for the reverse.".

constant: BootSource.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to the actual medium from
which the operating system is loaded, independently from what the
#$BootSequence-BIOSsetting is.".

constant: FloppyAccess.
isa: PossibleObservable.
comment: "This #$PossibleObservable refers to whether the
#$FloppyDiskDrive is actually accessed by the BIOS during boot-time
system test.".

constant: DetectionMessage.
isa: PossibleObservable.
comment: "This #$PossibleObservable is related to BIOS messages
concerning the autodetection of the #$HardDiskDrives.".
```

```
constant: InFloppy.
isa: PossibleObservable.
comment: "This #$PossibleObservable is related to the #$OSfloppyDisk
being inside the #$FloppyDiskDrive.".

;***** DEFINITIONS OF INSTANCES FOR $PossibleObservableValue COLLECTION ****

constant: BootTime.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed occured during boot-time, i.e., from the time the power is
turned on until the Operating System starts being loaded.".

constant: ComponentSpecific.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$ProblemContext #$PossibleObservable. It means that the fault being
diagnosed is identified to be related with a specific #$PCComponent,
e.g., the #$Monitor, #$MotherBoard, #$HardDisk e.t.c.".

constant: SingleBeep.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$SpeakerBeep #$PossibleObservable. It means that the #$Speaker
produced a single beep".

constant: RingingOrBuzzing.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$SpeakerBeep #$PossibleObservable. It means that the #$Speaker is
producing a ringing or buzzing sound.".

constant: ConsistentPattern.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$SpeakerBeep #$PossibleObservable. It means that the #$Speaker is
producing a consistent pattern (code) of beeps, e.g., one beep, then
two more.".

constant: Complete.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to some tests
performed by the BIOS, e.g., the memory test. It means that the
corresponding test is succesfully completed.".

constant: InComplete.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to some tests
performed by the BIOS, e.g., the memory test. It means that the
corresponding test is not succesfully completed.".

constant: CannotFind-Message.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to BIOS error
messages concerning the autodetection of IDE/ATAPI devices,
e.g. #$HardDiskDrive, #$CDROMdrive. This kind of messages indicate
that the BIOS cannot detect the corresponding device.".

constant: Auto.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue indicates that some
action/process/procedure is (set to be) done automatically.".

constant: Manual.
isa: PossibleObservableValue.
```

```
comment: "This #$PossibleObservableValue indicates that some
action/process/procedure is (set to be) done manually.".

constant: FloppyThenHard.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$BootSequence-BIOSsetting #$PossibleObservable. It indicates that
this setting is set to A:-C:.".

constant: HardThenFloppy.
isa: PossibleObservableValue.
comment: "This #$PossibleObservableValue is related to the
#$BootSequence-BIOSsetting #$PossibleObservable. It indicates that
this setting is set to C:-A:.".

;;; ********* IMPLEMENTATION OF KADS SYSTEMATIC DIAGNOSIS PSM ********

;;; The Task Structure for Systematic Diagnosis (pseudo-code) is:

;;; Systematic Diagnosis(+complaint,+possible observables,-hypothesis) by
;;;  select1(+complaint, -system model)
;;;  REPEAT
;;;   decompose(+system model, -hypothesis)
;;;   WHILE number of hypotheses > 1
;;;    select2(+possible observables, -variable value)
;;;    select3(+hypothesis, -norm)
;;;    compare(+variable value, +norm, -difference)
;;;   system model <- current decomposition level of system model
;;;  UNTIL confirm(+hypothesis), i.e. system model cannot be decomposed further

;;; The user interaction in CYC will be done from the SubL Interactor
;;; interface, as it is not possible to get any input/output
;;; interaction between the user and the SubL code from the ASK
;;; interface. The whole Task Structure will be implemented as a SubL
;;; function, 'systematic', which will be responsible for calling the
;;; appropriate SubL functions that will implement the corresponding
;;; inferences. In fact, the 'select1', 'select2', and 'select3'
;;; inferences will be implemented as FORWARD rules in the
;;; CYC KB. "Forward" means that, according to the results of 'Tests'
;;; that the user is asked to give, these rules automatically assert
;;; new facts in the KB. These facts describe which are the next
;;; #$PossibleObservables and Variables that must be tested ('select2'
;;; inference), what should be done according to the result ('select3'
;;; and 'compare' inferences), e.g., if another test for the same
;;; hypothesis should be performed or if the current hypothesis should
;;; be rejected or the current hypothesis must be decomposed further
;;; or if the faulty component was found ('confirm' inference).

;;; During the Systematic Diagnosis problem solving method (PSM) as
;;; well as during any other PSM, there are certain decisions/choices
;;; that must be done. According to the structure of the PSMs as
;;; Generic Task Models in KADS, these decisions/choices occur during
;;; the performance of specific Inferences. The implementation of
;;; these decisions/choices has to be declarative since this is the
;;; main principle in CYC. Therefore, the implementation of them will
;;; be in terms of FORWARD rules: the ANTECENDENT of each rule will be
;;; the conditions under which a decision is made ant the CONSEQUENT
;;; will be the knowledge that is becoming known to the system when
;;; this decision is made. Then, the newly added information will be
;;; used by the SubL code to guide the whole procedure. In the
;;; following, we will examine in detail which these decisions/choices
;;; are, when and where do they occur and how the are actually
;;; implemente as forward rules.

;;; The PSM starts with a SELECT inference. A general symptom is
;;; entered by the user and an appropriate system model is
```

```
;;; chosen. This inference is slightly changed for PC diagnosis. What
;;; is actually asked from the user is to distinguish three (3) major
;;; contexts of diagnosis: (i) Boot-time troubleshooting, (ii)
;;; Run-time troubleshooting and (iii) Component-specific
;;; troubleshooting. This categorisation is significant since
;;; completely different rules are applicable in each
;;; context. Although this contextual dependency of the rules could be
;;; implemented as different #$Microtheory contexts, this would make
;;; context shifting more complicated - any ASK operation would have
;;; to define the #$Microtheory. Instead, this dependency will be
;;; embedded in the antecedent part of the rules as an extra
;;; condition. The special predicate diagnosisContext (see
;;; #$SystematicDiagnosis microtheory) will be used, e.g.,
;;;
;;; (implies
;;;  (and
;;;   (diagnosisContext BootTime)
;;;   (...<more conditions>...)) ;end of antecedent
;;;  (<consequent>))


;;; Rule to assert a (#$diagnosisContext ...) assertion. This assertion
;;; is introduced as a "shorthand" for the assertion:
;;;
;;; (resultOfTest (TestFn PCSystem ConfirmSensorially ProblemContext) ?PROBLEM)
;;;
;;; It is used as a premise in every rule which is applicable to the
;;; corresponding diagnosis context, i.e., #$BootTime, #$RunTime or
;;; #$ComponentSecific.


Direction: forward.
F: (implies
    (resultOfTest (TestFn PCSystem ConfirmSensorially ProblemContext) ?PROBLEM)
    (diagnosisContext ?PROBLEM)).

;;; The 'decompose' inference in KADS Systematic Diagnosis PSM takes as input
;;; the current #$PCSubSystem (#$hypothesis PC_SUBSYSTEM) and decomposes it
;;; into its functional subsystems (#$functionalPartOf PC_SUBSYSTEM PART),
;;; generating new hypotheses (#$possibleHypotheses PART).

;;; **********************************
;;; ** Rules and Facts for Decompose **
;;; **********************************

;; Every PART which is a functional part of the hypothesis, HYP, is a
;; possible hypothesis.

direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (plausibleInference Decompose)
     (hypothesis ?HYP)
     (functionalPartOf ?HYP ?PART))
    (possibleHypotheses ?PART)).



F: (functionalPartOf PCSystem PowerSystem).
F: (functionalPartOf PCSystem VideoSystem).
F: (functionalPartOf PCSystem BIOSStartupSystem).
F: (functionalPartOf PCSystem MemorySystem).
F: (functionalPartOf PCSystem FloppySystem).
F: (functionalPartOf PCSystem HardDiskDrive).
F: (functionalPartOf PCSystem CDROMdrive).
```

```
F: (functionalPartOf PCSystem PlugAndPlaySystem).
F: (functionalPartOf PCSystem BootSystem).



F: (functionalPartOf PowerSystem PowerSocket).
F: (functionalPartOf PowerSystem PowerCable).
F: (functionalPartOf PowerSystem PowerProtectionDevice).
F: (functionalPartOf PowerSystem PowerSupply).


F: (functionalPartOf VideoSystem MotherBoard).
F: (functionalPartOf VideoSystem VideoCard).
F: (functionalPartOf VideoSystem Monitor).

F: (functionalPartOf BIOSStartupSystem MotherBoard).
F: (functionalPartOf BIOSStartupSystem VideoCard).

F: (functionalPartOf MemorySystem RAM).
F: (functionalPartOf MemorySystem MotherBoard).

F: (functionalPartOf FloppySystem FloppyDiskDrive).
F: (functionalPartOf FloppySystem BIOSsettings).


F: (functionalPartOf PlugAndPlaySystem ExpansionCard).
F: (functionalPartOf PlugAndPlaySystem MotherBoard).

F: (functionalPartOf BootSystem FloppyDiskDrive).
F: (functionalPartOf BootSystem OSfloppyDisk).
F: (functionalPartOf BootSystem HardDiskDrive).

;;; ******************************************************
;;; **                                                  **
;;; ** Facts and rules about PC subsystems and components **
;;; **                                                  **
;;; ******************************************************

;;; ******************************
;;; ** TEST(S) for the PCSystem **
;;; ******************************

Default Mt: PCDiagnosisMt.

Direction: forward.
F: (implies
    (hypothesis PCSystem) ;if diagnosis just started, ask for the context
    (and
     (possibleTest (TestFn PCSystem ConfirmSensorially ProblemContext))
     (possibleResultOfTest
      (TestFn PCSystem ConfirmSensorially ProblemContext) BootTime NotNormal)
     (possibleResultOfTest
      (TestFn PCSystem ConfirmSensorially ProblemContext) RunTime  NotNormal)
     (possibleResultOfTest
      (TestFn PCSystem ConfirmSensorially ProblemContext)
                                                 ComponentSpecific NotNormal))).

;; *********************************************
;; ** DECOMPOSITION knowledge for the PCSystem **
;; *********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
```

```
     (hypothesis PCSystem)
     (plausibleInference Decompose))
    (and
     (testFirst PowerSystem)
     (testAfter PowerSystem VideoSystem)
     (testAfter VideoSystem BIOSStartupSystem)
     (testAfter BIOSStartupSystem MemorySystem)
     (testAfter MemorySystem FloppySystem)
     (testAfter FloppySystem HardDiskDrive)
     (testAfter HardDiskDrive CDROMdrive)
     (testAfter CDROMdrive PlugAndPlaySystem)
     (testAfter PlugAndPlaySystem BootSystem))).


;; ********************************
;; ** TEST(S) for the PowerSystem **
;; ********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSystem)) ;
    (and
     (possibleTest (TestFn PowerSystem ConfirmSensorially ElectricPower))
     (possibleResultOfTest
      (TestFn PowerSystem ConfirmSensorially ElectricPower) Yes Normal)
     (possibleResultOfTest
      (TestFn PowerSystem ConfirmSensorially ElectricPower) No NotNormal))).

;; ***********************************************
;; ** DECOMPOSITION knowledge for the PowerSystem **
;; ** ElectricPower=No                          **
;; ***********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSystem)
     (resultOfTest (TestFn PowerSystem ConfirmSensorially ElectricPower) No)
     (plausibleInference Decompose))
    (and
     (testFirst PowerSocket)
     (testAfter PowerSocket PowerProtectionDevice)
     (testAfter PowerProtectionDevice PowerCable)
     (testAfter PowerCable PowerSupply))).

;; ********************************
;; ** TEST(S) for the PowerSocket **
;; ********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSocket)) ;
    (and
     (possibleTest (TestFn PowerSocket CheckIndependently ElectricPower))
     (possibleResultOfTest
      (TestFn PowerSocket CheckIndependently ElectricPower) Yes Normal)
     (possibleResultOfTest
      (TestFn PowerSocket CheckIndependently ElectricPower) No NotNormal))).

;;*******************************************
```

```
;; ** TEST(S) for the PowerProtectionDevice **
;;*******************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerProtectionDevice))
    (and
     (possibleTest (TestFn PowerProtectionDevice Remove ElectricPower))
     (possibleResultOfTest
      (TestFn PowerProtectionDevice Remove ElectricPower) Yes NotNormal)
     (possibleResultOfTest
      (TestFn PowerProtectionDevice Remove ElectricPower) No Normal))).


;;*********************************
;; ** TEST(S) for the PowerCable **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerCable)) ;
    (and
     (possibleTest (TestFn PowerCable Replace ElectricPower))
     (possibleResultOfTest
      (TestFn PowerCable Replace ElectricPower) Yes NotNormal)
     (possibleResultOfTest
      (TestFn PowerCable Replace ElectricPower) No Normal))).

;;*********************************
;;** TEST(S) for the PowerSupply **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSupply))
    (and
     (possibleTest (TestFn PowerSupply ConfirmSensorially VoltageCorrect))
     (possibleResultOfTest
      (TestFn PowerSupply ConfirmSensorially VoltageCorrect) Yes NotNormal)
     (possibleResultOfTest
      (TestFn PowerSupply ConfirmSensorially VoltageCorrect) No Insufficient))).

;;*********************************
;;** TEST(S) for the PowerSupply **
;;** VoltageCorrect=No           **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PowerSupply)
     (resultOfTest (TestFn PowerSupply ConfirmSensorially VoltageCorrect) No)) ;
    (and
     (possibleTest (TestFn PowerSupply ChangeVoltage ElectricPower))
     (possibleResultOfTest
      (TestFn PowerSupply ChangeVoltage ElectricPower) Yes Normal)
     (possibleResultOfTest
      (TestFn PowerSupply ChangeVoltage ElectricPower) No NotNormal))).
```

```
;;*********************************
;;** TEST(S) for the VideoSystem **
;;*********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem))
    (and
     (possibleTest (TestFn VideoSystem ConfirmSensorially VideoSignal))
     (possibleResultOfTest
      (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes Insufficient)
     (possibleResultOfTest
      (TestFn VideoSystem ConfirmSensorially VideoSignal) No NotNormal))).

;; ***********************************************
;; ** DECOMPOSITION knowledge for the VideoSystem **
;; ** VideoSignal=No                             **
;; ***********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No)
     (plausibleInference Decompose))
    (and
     (testFirst Monitor)
     (testAfter Monitor MotherBoard)
     (testAfter MotherBoard Speaker))).

;;********************************
;;** TEST(S) for the VideoSystem **
;;** VideoSignal=Yes            **
;;********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes))
    (and
     (possibleTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage))
    (possibleResultOfTest
     (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes Insufficient)
    (possibleResultOfTest
     (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No NotNormal))).

;; *****************************************
;; ** TEST(S) for the VideoSystem          **
;; ** VideoSignal=Yes, VideoBIOSMessage=Yes **
;; *****************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes))
    (and
     (possibleTest (TestFn VideoSystem ConfirmSensorially BootContinues))
     (possibleResultOfTest
```

```
       (TestFn VideoSystem ConfirmSensorially BootContinues) Yes Normal)
      (possibleResultOfTest
       (TestFn VideoSystem ConfirmSensorially BootContinues) No NotNormal))).

;; *************************************************************
;; ** DECOMPOSITION knowledge for the VideoSystem             **
;; ** VideoSignal=Yes, VideoBIOSMessage=Yes, BootContinues=No **
;; *************************************************************


Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially BootContinues) No)
     (plausibleInference Decompose))
    (and
     (testFirst VideoCard)
     (testAfter VideoCard MotherBoard))).

;; **************************************************
;; ** DECOMPOSITION knowledge for the VideoSystem  **
;; ** VideoSignal=Yes, VideoBIOSMessage=No         **
;; **************************************************


Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoSystem)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
     (plausibleInference Decompose))
    (and
     (testFirst MotherBoard)
     (testAfter MotherBoard VideoCard))).


;; *****************************
;; ** TEST(S) for the Monitor **
;; ** VideoSignal=No          **
;; *****************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis Monitor)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No))
    (and
     (possibleTest (TestFn Monitor CheckIndependently VideoSignal))
     (possibleResultOfTest
      (TestFn Monitor CheckIndependently VideoSignal) Yes Normal)
     (possibleResultOfTest
      (TestFn Monitor CheckIndependently VideoSignal) No NotNormal))).

;; ******************************************
;; ** TEST(S) for the MotherBoard          **
;; ** VideoSignal=No, Monitor_Working =Yes **
;; ******************************************

Direction: forward.
```

```
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No)
     (resultOfTest (TestFn Monitor CheckIndependently VideoSignal) Yes))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) SingleBeep NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep)
                                          ConsistentPattern NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep)
                                          RingingOrBuzzing NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient))).

;; **********************************************************
;; ** TEST(S) for the MotherBoard                          **
;; ** VideoSignal=No, Monitor_Working =Yes, SpeakerBeep=No **
;; **********************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) No)
     (resultOfTest (TestFn Monitor CheckIndependently VideoSignal) Yes)
     (resultOfTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No))
    (and
     (possibleTest (TestFn Speaker CheckIndependently SpeakerBeep))
     (possibleResultOfTest
      (TestFn Speaker CheckIndependently SpeakerBeep) Yes NotNormal)
     (possibleResultOfTest
      (TestFn Speaker CheckIndependently SpeakerBeep) No Distinguishing))).

;;; Important notice: Although the PCSusbSystem related to the Test is
;;; the Speaker, however, the PCSubSystem hypothesised as faulty is
;;; the MotherBoard. Therefore, the Result Type is defined by what it
;;; indicates about the MotherBoard and not the Speaker itself. This
;;; rather peculiar situation is due to the controlling function of
;;; the Speaker. This means that the Speaker functions as a control
;;; device for the MotherBoard and we must make sure that this device
;;; is working properly. If it does, then the lack of any sound is due
;;; to the MotherBoard and we can deduce that it is faulty. Otherwise,
;;; we cannot deduce anything before we are certain that the Speaker
;;; is working properly.


;; ******************************************
;; ** TEST(S) for the MotherBoard          **
;; ** VideoSignal=Yes, VideoBIOSMessage=No **
;; ******************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
```

```
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) ConsistentPattern
                                                               Insufficient))).

;; *************************************************************************
;; ** TEST(S) for the MotherBoard                                         **
;; ** VideoSignal=Yes, VideoBIOSMessage=No, SpeakerBeep=ConsistentPattern **
;; *************************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
     (resultOfTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep)
                                                            ConsistentPattern))
    (and
     (possibleTest (TestFn MotherBoard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; **********************************************************
;; ** TEST(S) for the MotherBoard                          **
;; ** VideoSignal=Yes, VideoBIOSMessage=No, SpeakerBeep=No **
;; **********************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) No)
     (resultOfTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No))
    (and
     (possibleTest (TestFn MotherBoard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; *********************************
;; ** TEST(S) for the MotherBoard **
;; ** StartupScreen=No            **
;; *********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest
       (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially SpeakerBeep))
     (possibleResultOfTest
```

```
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) Yes NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No Insufficient))).

;; **************************************
;; ** TEST(S) for the MotherBoard      **
;; ** StartupScreen=No, SpeakerBeep=No **
;; **************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest
      (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No)
     (resultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No))
    (and
     (possibleTest (TestFn MotherBoard ConfirmSensorially ErrorMessage))
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially ErrorMessage) Yes NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard ConfirmSensorially ErrorMessage) No Insufficient))).

;; *******************************************************
;; ** TEST(S) for the MotherBoard                       **
;; ** StartupScreen=No, SpeakerBeep=No, ErrorMessage=No **
;; *******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MotherBoard)
     (resultOfTest
      (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No)
     (resultOfTest
      (TestFn MotherBoard ConfirmSensorially SpeakerBeep) No)
     (resultOfTest
      (TestFn MotherBoard ConfirmSensorially ErrorMessage) No))
    (and
     (possibleTest (TestFn MotherBoard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn MotherBoard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).



;; *************************************************************
;; ** TEST(S) for the VideoCard                               **
;; ** VideoSignal=Yes, VideoBIOSMessage=Yes, BootContinues=No **
;; *************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis VideoCard)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoSignal) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially VideoBIOSMessage) Yes)
     (resultOfTest (TestFn VideoSystem ConfirmSensorially BootContinues) No))
    (and
```

```
      (possibleTest (TestFn VideoCard TroubleshootComponent ComponentProblem))
      (possibleResultOfTest
       (TestFn VideoCard TroubleshootComponent ComponentProblem) Yes NotNormal)
      (possibleResultOfTest
       (TestFn VideoCard TroubleshootComponent ComponentProblem) No
                                                              Distinguishing))).

;;; Important notice: The '#$TroubleshootComponent' TestAction and the
;;; '#$ComponentProblem' PossibleObservable are too general and
;;; complex to be of actual use. They are used here as artificial
;;; "terminating points" of the Systematic Diagnosis procedure at the
;;; level of #$PCComponent. In a
;;; complete PC faults diagnosis expert system, more elaborate
;;; 'Test(s)' should follow to troubleshoot a specific #$PCComponent
;;; (here the #$VideoCard), instead of the user having to know how to
;;; test the specific #$PCComponent. However, these 'Tests' are so
;;; elaborate and complicated that are beyond the scope of this
;;; implementation. Such 'Test(s)' could be identifying a beep code
;;; according to the specific version of BIOS (American Megatrends
;;; Inc., Pheonix or Other) or interpreting an error message (there
;;; are 120 error messages documented in the PCGuide Troubleshoot
;;; expert that was used as a knowledge acquisition source).

;; *************************************
;; ** TEST(S) for the BIOSStartupSystem **
;; ** PowerSystem=ok, VideoSystem=ok    **
;; *************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BIOSStartupSystem))
    (and
     (possibleTest (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen))
     (possibleResultOfTest
      (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) Yes Normal)
     (possibleResultOfTest
     (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No NotNormal))).

;; ****************************************************
;; ** DECOMPOSITION knowledge for the BIOSStartupSystem **
;; ** StartupScreen=No                                 **
;; ****************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BIOSStartupSystem)
     (resultOfTest
       (TestFn BIOSStartupSystem ConfirmSensorially StartupScreen) No)
     (plausibleInference Decompose))
    (and
     (testFirst MotherBoard)
     (testAfter MotherBoard VideoCard))).

;; ***************************************************
;; ** TEST(S) for the MemorySystem                    **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ***************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
```

```
     (hypothesis MemorySystem))
    (and
     (possibleTest (TestFn MemorySystem ConfirmSensorially MemoryTest))
     (possibleResultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) Complete Normal)
     (possibleResultOfTest
     (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete NotNormal))).

;; ***********************************************
;; ** DECOMPOSITION knowledge for the MemorySystem **
;; ** MemoryTest=Incomplete                        **
;; ***********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis MemorySystem)
     (resultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete)
     (plausibleInference Decompose))
    (and
     (testFirst RAM)
     (testAfter RAM MotherBoard))).

;; **************************
;; ** TEST(S) for the RAM   **
;; ** MemoryTest=Incomplete **
;; **************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis RAM)
     (resultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete))
    (and
     (possibleTest (TestFn RAM ConfirmSensorially ErrorMessage))
     (possibleResultOfTest
      (TestFn RAM ConfirmSensorially ErrorMessage) Yes NotNormal)
     (possibleResultOfTest
      (TestFn RAM ConfirmSensorially ErrorMessage) No Insufficient))).

;; *****************************************
;; ** TEST(S) for the RAM                   **
;; ** MemoryTest=Incomplete, ErrorMessage=No **
;; *****************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis RAM)
     (resultOfTest
      (TestFn MemorySystem ConfirmSensorially MemoryTest) InComplete)
     (resultOfTest
      (TestFn RAM ConfirmSensorially ErrorMessage) No))
    (and
     (possibleTest (TestFn RAM TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn RAM TroubleshootComponent ComponentProblem) Yes NotNormal)
     (possibleResultOfTest
      (TestFn RAM TroubleshootComponent ComponentProblem) No
                                                            Distinguishing))).
```

```
;; ****************************************************
;; ** TEST(S) for the FloppySystem                   **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok                                **
;; ****************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis FloppySystem))
    (and
     (possibleTest (TestFn FloppySystem ConfirmSensorially FloppyAccess))
     (possibleResultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) Yes Normal)
     (possibleResultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) No NotNormal))).

;; ************************************************
;; ** DECOMPOSITION knowledge for the FloppySystem **
;; ** FloppyAccess=No                            **
;; ************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis FloppySystem)
     (resultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) No)
     (plausibleInference Decompose))
    (and
     (testFirst FloppyDiskDrive)
     (testAfter FloppyDiskDrive BIOSsettings))).

;; ************************************
;; ** TEST(S) for the FloppyDiskDrive **
;; ** FloppyAccess=No                **
;; ************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis FloppyDiskDrive)
     (resultOfTest
      (TestFn FloppySystem ConfirmSensorially FloppyAccess) No))
    (and
     (possibleTest (TestFn FloppyDiskDrive ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn FloppyDiskDrive ConfirmSensorially BootContinues) Yes
                                                            Distinguishing)
     (possibleResultOfTest
      (TestFn FloppyDiskDrive ConfirmSensorially BootContinues) No NotNormal))).


;; ***************************************************************
;; ** TEST(S) for the OSfloppyDisk                              **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=Hard/None **
;; ***************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis OSfloppyDisk)
```

```
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
     (or
       (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource)
                                             (ObservableValueFn HardDiskDrive))
       (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource) None)))
    (and
     (possibleTest (TestFn OSfloppyDisk ConfirmSensorially InFloppy))
     (possibleResultOfTest
      (TestFn OSfloppyDisk ConfirmSensorially InFloppy) Yes Insufficient)
     (possibleResultOfTest
      (TestFn OSfloppyDisk ConfirmSensorially InFloppy) No NotNormal))).

;; *******************************************************************
;; ** TEST(S) for the OSfloppyDisk                                  **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=Hard/None **
;; ** InFloppy=Yes                                                  **
;; *******************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis OSfloppyDisk)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
     (or
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource)
                                             (ObservableValueFn HardDiskDrive))
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource) None))
     (resultOfTest
      (TestFn OSfloppyDisk ConfirmSensorially InFloppy) Yes))
    (and
     (possibleTest
      (TestFn OSfloppyDisk TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn OSfloppyDisk TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn OSfloppyDisk TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; ******************************************************
;; ** TEST(S) for the HardDiskDrive                    **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok, FloppySystem=ok                 **
;; ******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis HardDiskDrive))
    (and
     (possibleTest (TestFn HardDiskDrive ConfirmSensorially DetectionMessage))
     (possibleResultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage) Yes Normal)
     (possibleResultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage) No Insufficient)
     (possibleResultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage)
                                               CannotFind-Message NotNormal))).

;; ***********************************
```

```
;; ** TEST(S) for the HardDiskDrive **
;; ** DetectionMessage=No            **
;; ************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis HardDiskDrive)
     (resultOfTest
      (TestFn HardDiskDrive ConfirmSensorially DetectionMessage) No))
    (and
     (possibleTest
      (TestFn HardDiskDrive CheckIndependently AutoDetection-BIOSsetting))
     (possibleResultOfTest
      (TestFn HardDiskDrive CheckIndependently AutoDetection-BIOSsetting)
                                                                 Manual Normal)
     (possibleResultOfTest
     (TestFn HardDiskDrive CheckIndependently AutoDetection-BIOSsetting)
                                                             Auto NotNormal))).

;; *********************************************************************
;; ** TEST(S) for the HardDiskDrive                                  **
;; ** BootSequence-BIOSsetting=HardThenFloppy, BootSource=Floppy/None **
;; *********************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis HardDiskDrive)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                HardThenFloppy)
     (or
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource)
                                           (ObservableValueFn FloppyDiskDrive))
      (resultOfTest (TestFn BootSystem ConfirmSensorially BootSource) None)))
    (and
     (possibleTest
      (TestFn HardDiskDrive TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn HardDiskDrive TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn HardDiskDrive TroubleshootComponent ComponentProblem) No
                                                             Normal))).

;; ******************************************************
;; ** TEST(S) for the CDROMdrive                       **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok, FloppySystem=ok, HardDisk=ok    **
;; ******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis CDROMdrive))
    (and
     (possibleTest (TestFn CDROMdrive ConfirmSensorially DetectionMessage))
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) Yes Normal)
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) No Insufficient)
     (possibleResultOfTest
```

```
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) CannotFind-Message
                                                                 NotNormal))).

;; ***********************************
;; ** TEST(S) for the CDROMdrive **
;; ** DetectionMessage=No           **
;; ***********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis CDROMdrive)
     (resultOfTest
      (TestFn CDROMdrive ConfirmSensorially DetectionMessage) No))
    (and
     (possibleTest (TestFn CDROMdrive ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn CDROMdrive ConfirmSensorially BootContinues) Yes Normal)
     (possibleResultOfTest
     (TestFn CDROMdrive ConfirmSensorially BootContinues) No NotNormal))).

;; ******************************************************
;; ** TEST(S) for the PlugAndPlaySystem                **
;; ** PowerSystem=ok, VideoSystem=ok, StartupSystem=ok **
;; ** MemorySystem=ok, FloppySystem=ok, HardDisk=ok    **
;; ** CDROMdrive=ok                                    **
;; ******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PlugAndPlaySystem))
    (and
     (possibleTest (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues))
     (possibleResultOfTest
      (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) Yes Normal)
     (possibleResultOfTest
     (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) No NotNormal))).

;; *******************************************************
;; ** DECOMPOSITION knowledge for the PlugAndPlaySystem **
;; ** BootContinues=No                                  **
;; *******************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis PlugAndPlaySystem)
     (resultOfTest
      (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) No)
     (plausibleInference Decompose))
    (and
     (testFirst ExpansionCard)
     (testAfter ExpansionCard MotherBoard))).

;; ***********************************
;; ** TEST(S) for the ExpansionCard **
;; ** BootContinues=No              **
;; ***********************************

Direction: forward.
F: (implies
    (and
```

```
     (diagnosisContext BootTime)
     (hypothesis ExpansionCard)
     (resultOfTest
      (TestFn PlugAndPlaySystem ConfirmSensorially BootContinues) No))
    (and
     (possibleTest
      (TestFn ExpansionCard TroubleshootComponent ComponentProblem))
     (possibleResultOfTest
      (TestFn ExpansionCard TroubleshootComponent ComponentProblem) Yes
                                                                     NotNormal)
     (possibleResultOfTest
      (TestFn ExpansionCard TroubleshootComponent ComponentProblem) No
                                                             Distinguishing))).

;; *********************************
;; ** TEST(S) for the BootSystem   **
;; ** Everything else=ok           **
;; *********************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem))
    (and
     (possibleTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting))
     (possibleResultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                   FloppyThenHard Insufficient)
     (possibleResultOfTest
     (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                HardThenFloppy Insufficient))).

;; *********************************************
;; ** TEST(S) for the BootSystem               **
;; ** BootSequence-BIOSsetting=FloppyThenHard  **
;; *********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                               FloppyThenHard))
    (and
     (possibleTest (TestFn BootSystem ConfirmSensorially BootSource))
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                    (ObservableValueFn FloppyDiskDrive) Normal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                   (ObservableValueFn HardDiskDrive) NotNormal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None NotNormal))).

;; ************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem              **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=Hard **
;; ************************************************************

Direction: forward.
F: (implies
    (and
```

```
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
     (resultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                             (ObservableValueFn HardDiskDrive))
     (plausibleInference Decompose))
    (and
     (testFirst OSfloppyDisk)
     (testAfter  OSfloppyDisk FloppyDiskDrive))).


;; **************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem               **
;; ** BootSequence-BIOSsetting=FloppyThenHard, BootSource=None **
;; **************************************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                                FloppyThenHard)
     (resultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None)
     (plausibleInference Decompose))
    (and
     (testFirst OSfloppyDisk)
     (testAfter OSfloppyDisk FloppyDiskDrive))).

;; **********************************************
;; ** TEST(S) for the BootSystem               **
;; ** BootSequence-BIOSsetting=HardThenFloppy  **
;; **********************************************

Direction: forward.
F: (implies
    (and
     (diagnosisContext BootTime)
     (hypothesis BootSystem)
     (resultOfTest
      (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                               HardThenFloppy))
    (and
     (possibleTest (TestFn BootSystem ConfirmSensorially BootSource))
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                    (ObservableValueFn HardDiskDrive) Normal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource)
                                (ObservableValueFn FloppyDiskDrive) NotNormal)
     (possibleResultOfTest
      (TestFn BootSystem ConfirmSensorially BootSource) None NotNormal))).

;; ****************************************************************
;; ** DECOMPOSITION knowledge for the BootSystem                 **
;; ** BootSequence-BIOSsetting=HardThenFloppy, BootSource=Floppy **
;; ****************************************************************

Direction: forward.
F: (implies
    (and
```

```
      (diagnosisContext BootTime)
      (hypothesis BootSystem)
      (resultOfTest
       (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                             HardThenFloppy)
      (resultOfTest
       (TestFn BootSystem ConfirmSensorially BootSource)
                                     (ObservableValueFn FloppyDiskDrive))
      (plausibleInference Decompose))
    (testFirst HardDiskDrive)).

;; ***********************************************************
;; ** DECOMPOSITION knowledge for the BootSystem             **
;; ** BootSequence-BIOSsetting=HardThenFloppy, BootSource=None **
;; ***********************************************************

Direction: forward.
F: (implies
     (and
      (diagnosisContext BootTime)
      (hypothesis BootSystem)
      (resultOfTest
       (TestFn BootSystem CheckIndependently BootSequence-BIOSsetting)
                                                             HardThenFloppy)
      (resultOfTest
       (TestFn BootSystem ConfirmSensorially BootSource) None)
      (plausibleInference Decompose))
     (testFirst HardDiskDrive)).

**********************************************************************
**********************************************************************
                END OF pc_diagnosisKE.txt
**********************************************************************
**********************************************************************
```

Appendix E

The CYC SubL Code for PC/Automobile Domains

```
;;; ******** FUNCTION DEFINITIONS ********

;;; ******** GLOBAL VARIABLES ****************

(csetq *use-local-queue?* NIL) ;CYC variable

(defvar *defaultMt* nil) ;the microtheory for the fi-ask function

(defvar *test* nil) ; the current Test (needed by the 'menu' function)

(defvar *results* nil) ; a list of the Possible Results of the current Test
                       ; (needed by the 'menu' function)

(defvar *no_of_choices* 0) ;the number of Possible Results of the current Test
                           ; (needed by the 'menu' function)

(defvar *system* nil) ;the system being diagnosed (see 'systematic')

;;; The global variable *terms* is used to record in a list - and in
;;; parallel with what the 'format' expressions print to the CYC SubL
;;; Interactor panel - the CYC terms involved in each step of the
;;; Systematic diagnosis. This variable is returned to the Interactor
;;; panel as the 'Results' of the 'Last Form Evaluated' (see the CYC
;;; SubL Interactor panel). The Interactor panel converts any CYC
;;; term, i.e. symbols starting with '#$' (hash-dollar), as an HTML
;;; link to this term in the CYC KB. This way, the user can browse to
;;; any of the CYC terms appearing on the screen.

(defvar *terms* nil)


;;; ********************************************
;;; ******** SubL CODE FOR SYSTEMATIC ************
;;; ********************************************

;;; Function : SYSTEMATIC
;;; Arguments: The system that is going to be diagnosed. For the
;;;           moment, any one of the symbols '#$PCSystem' or
;;;           '#$AutomobileSystem'.
;;; Result   : Implements the Inference Structure for the Systematic
;;;            Diagnosis problem solving method of KADS applied in PC
;;;            and Automobile faults diagnosis.
;;; Remarks  :
```

```
(define systematic (system)
  (pcond
   ((cnot (cor (eql system '#$PCSystem) (eql system '#$AutomobileSystem)))
     (error "SYSTEMATIC:argument must be '#$PCSystem' or '#$AutomobileSystem"))
  (t (pcond
      ((eql system '#$PCSystem) (csetf *defaultMt* '#$PCDiagnosisMt))
      ((eql system '#$AutomobileSystem)
                       (csetf *defaultMt* '#$AutomobileDiagnosisMt))
     ) ;end of pcond
     (pcond
      ;; if there isn't any hypothesis then start diagnosis
      ((cnot (fi-ask '(#$hypothesis ?HYP) *defaultMt*))
        (csetf *terms* nil)
        (csetf *system* system)
        (sd-select1 system))
      (t (sd-compare)) ;end of innermost 't' clause
     )) ;end of innermost 'pcond' and outermost 't' clause
 ) ;end of outermost 'pcond'
)

;;; *********************************************
;;; ******** SubL CODE FOR SD-COMPARE * *************
;;; *********************************************

;;; Function : SD-COMPARE
;;; Arguments: None
;;; Result   : Calls the appropriate Systematic Diagnosis Inference
;;; Remarks  : The *test* global variable holds the last 'Test' that
;;;            was performed. It is used to retrieve the result of
;;;            this 'Test' and its type. According to the 'ResultType'
;;;            of the 'Test', the following decisions may be made:
;;;
;;;            RESULT_TYPE             DECISION
;;;
;;;             Normal                  Reject hypothesis; backtrack
;;;             NotNormal               Decompose/Confirm hypothesis
;;;             Insufficient            Perform another 'Test' (Select2-3)
;;;             Distinguishing          Reject hypothesis, backtrack
;;;                                     and confirm next hypothesis.

(define sd-compare ()
 (csetf *terms* nil)
;; get the 'ResultType' for the most recently performed 'Test'
 (csetq ask-result
   (fi-ask (list '#$and
            (list '#$resultOfTest *test* '?RES)
            (list '#$possibleResultOfTest *test* '?RES '?TYPE)) *defaultMt*))
 (format t "LAST TEST: ~S,      ~%RESULT: ~S," *test* (get-ask-binding
              (first ask-result) 1))
 (csetf *terms* (cons (cons 'LAST (cons 'TEST: *test*)) *terms*))

 (csetq result (get-ask-binding (first ask-result) 1))
 (csetq result-type (get-ask-binding (first ask-result) 2))

;; according to the 'ResultType' value, decide the next inference
 (pcond
  ((eql result-type '#$NotNormal)
     (format t " RESULT TYPE: #$NotNormal, INFERENCE: Confirm/Decompose")
     (csetf *terms* (cons (list 'RESULT: result 'RESULT 'TYPE:
                             result-type 'INFERENCE:CONFIRM) *terms*))
     (sd-confirm))
  ((eql result-type '#$Insufficient)
     (format t " RESULT TYPE: #$Insufficient, INFERENCE: Select2-3")
     (csetf *terms* (cons (list 'RESULT: result 'RESULT 'TYPE:
                             result-type 'INFERENCE:SELECT2-3) *terms*))
     (sd-select2-3))
```

```
   ((cor (eql result-type '#$Normal) (eql result-type '#$Distinguishing))
      (format t " RESULT TYPE: ~S, INFERENCE: NewHypothesis" result-type)
      (csetf *terms* (cons (list 'RESULT: result 'RESULT 'TYPE:
                           result-type 'INFERENCE:New_Hypothesis) *terms*))
      (sd-new-hypothesis result-type))
   (t (format t " RESULT TYPE: UNKNOWN!, INFERENCE: Diagnosis interrupted"))
  ) ;end of 'pcond'
 )

;;; *********************************************
;;; ******** SubL CODE FOR SD-CONFIRM ************
;;; *********************************************

;;; Function : SD-CONFIRM
;;; Arguments: None
;;; Result   : If the current 'hypothesis' is a 'SUbSystem', it
;;;            is decomposed by calling function 'sd-decompose';
;;;            otherwise, it is a Component and the diagnosis
;;;            terminates reporting this 'Component' as faulty.
;;; Remarks  :

(define sd-confirm ()
 (csetq ask-result
  (fi-ask '(#$and
            (#$hypothesis ?HYP)
            (#$isa ?HYP #$Component)) *defaultMt*))
 (pcond
  (ask-result
   (format t "~%~% DIAGNOSIS ENDED. FAULTY COMPONENT: ~S"
      (get-ask-binding (first ask-result) 1))
   (csetf *terms* (cons (list 'DIAGNOSIS 'ENDED. 'FAULTY 'COMPONENT:
                         (get-ask-binding (first ask-result) 1)) *terms*))
   (reverse *terms*))
  (t (format t " Decomposing...")
     (safe-fi :assert '(#$plausibleInference #$Decompose) *defaultMt*)
     (sd-decompose)
     (sd-select2-3))
 )
)

;;; *********************************************
;;; ******** SubL CODE FOR SD-NEW-HYPOTHESIS *****
;;; *********************************************

;;; Function: SD-NEW-HYPOTHESIS
;;; Arguments: 1. A #$ResultType
;;; Results:
;;; Remarks: This function is called by 'sd-confirm' with two types of
;;;          results, #$Normal and #$Distinguishing. If the result is
;;;          #$Normal, the function just changes the current
;;;          'hypothesis' to the next one (if one exists) and performs
;;;          the appropriate 'Test' by calling 'sd-select2-3'. If the result is
;;;          #$Distinguishing, then it changes the current
;;;          'hypothesis' as before, but doesn't perform any 'Test', as
;;;          the last 'Test' performed indicates that the next
;;;          'hypothesis' is faulty. Therefore, it calls the
;;;          'sd-confirm' function.

(define sd-new-hypothesis (result-type)

;; get the next hypothesis from the 'testAfter' assertions, if anyone
;; is left
 (csetq ask-result
       (fi-ask '(#$and
                 (#$hypothesis ?HYP)
                 (#$testAfter ?HYP ?NEW_HYP)) *defaultMt*))
```

```
 (pcond
  ((null ask-result) (format t "~% ~%I'M SORRY. THERE IS NO ALTERNATIVE
                        SYSTEM TO BE CONSIDERED. DIAGNOSIS FAILED"))
  (t (csetq hypothesis (get-ask-binding (first ask-result) 1))
     (csetq new_hypothesis (get-ask-binding (first ask-result) 2))
     (fi-unassert (list '#$hypothesis hypothesis) *defaultMt*)
     (fi-unassert (list '#$testAfter hypothesis new_hypothesis) *defaultMt*)
     (safe-fi :assert (list '#$hypothesis new_hypothesis) *defaultMt*)
     (pcond
      ((eql result-type '#$Normal) (sd-select2-3))
      ((eql result-type '#$Distinguishing) (sd-confirm))
     )) ;end of inner 'pcond' and 't' clause
 ) ;end of 'pcond'
)


;;; *********************************************
;;; ******** SubL CODE FOR SD-SELECT1********
;;; *********************************************

;;; Function: SD-SELECT1
;;; Arguments: The system that is going to be diagnosed. For the
;;;           moment, any one of the symbols '#$PCSystem' or
;;;           '#$AutomobileSystem'.
;;; Results: Asserts the '(#$hypothesis system)' fact to start the
;;;          Systematic Diagnosis problem solving method
;;; Remarks:

(define sd-select1 (system)
 (safe-fi :assert (list '#$hypothesis system) *defaultMt*)
 (sd-select2-3) ; ask the user what the general complaint is
)

;;; *********************************************
;;; ******** SubL CODE FOR SD-SELECT2-3 *************
;;; *********************************************

;;; Function : SD-SELECT2-3
;;; Arguments: None
;;; Result   :
;;; Remarks  : According to the situation, there may be a lot of
;;; 'possibleTest' assertions in the KB, but only one of them is the
;;; one that must be performed next. This Test is distinguished by the
;;; fact  that it is the only one that doesn't have a corresponding
;;; 'resultOfTest' assertion as it is not yet carried out. The
;;; 'sd-select2-3' function must therefore find this Test and pass it and
;;; its possible results ('possibleresultOfTest' assertions) to the
;;; 'get-test-result' function.


(define sd-select2-3 ()

;;get the current 'hypothesis'
 (csetq hypothesis (fi-ask '(#$hypothesis ?HYP) *defaultMt*))
 (csetq hypothesis (get-ask-binding (first hypothesis) 1))
 (format t "~%~%HYPOTHESIS: ~S" hypothesis)
 (csetf *terms* (cons (list  'HYPOTHESIS: hypothesis) *terms*))

;; Get all possible tests
 (csetq possible-tests
   (fi-ask '(#$possibleTest ?TEST) *defaultMt*))

;; keep the one that doesn't have a corresponding 'resultOfTest' assertion
 (cdo
  ((test
     (get-ask-binding (first possible-tests) 1)
```

```
      (get-ask-binding (first possible-tests) 1))
     ) ; end of variables
    ((cnot (fi-ask (list '#$resultOfTest test '?R) *defaultMt*)) t) ;exit condition
    (csetq possible-tests (rest possible-tests))
   )
   (csetq test (get-ask-binding (first possible-tests) 1))
   (csetf *test* test)

  ;; Get the possible results for this test
   (csetq possible-results
     (fi-ask
      (list
       '#$possibleResultOfTest (list '#$TestFn (second test) (third test)
                                    (fourth test)) '?VAL '?TYPE) *defaultMt*))
   (get-test-result test possible-results)
  )

  ;;; *********************************************
  ;;; ******** SubL CODE FOR GET-TEST-RESULT *****
  ;;; *********************************************

  ;;; Function : GET-TEST-RESULT
  ;;; Arguments: 1. A 'Test' structure, which is a list of the form:
  ;;;
  ;;; (TestFn SUBSYSTEM TEST_ACTION POSSIBLE_OBSERVABLE)
  ;;;
  ;;;            2. A list of the form:
  ;;;
  ;;; (
  ;;;  ((?VAL . RESULT_1) (?TYPE . TYPE_1))
  ;;;      ...
  ;;;  ((?VAL . RESULT_n) (?TYPE . TYPE_n))
  ;;; )
  ;;;
  ;;; Result   : An 'resultOfTest' Assertion in the KB with the actual result of
  ;;;            the test.
  ;;; Remarks  : For the moment, it is the 'menu' function that performs
  ;;;            the actual task as there in no way to get input from
  ;;;            the user when in the SubL interactor.

  (define get-test-result (test possible-results)
   (present-test-parameters test)
   (present-test-results possible-results)
   (format t "~%~%Please, type '(menu [number_of_result])'")
  ; (input-test-result no_of_choices)
   (reverse *terms*) ;return as RESULT a list of all CYC terms appearing on the screen

  ;;; For the moment, we don't know how to interact with the user when in the
  ;;; SubL Interactor interface. Therefore, the user must give the command
  ;;; '(menu <number_of_result>)' to interact with the SubL code.
  )


  ;;; **********************************************
  ;;; ******** SubL CODE FOR PRESENT-TEST-PARAMETERS*
  ;;; **********************************************

  ;;; Function : PRESENT-TEST-PARAMETERS
  ;;; Arguments: A 'Test' structure, which is a list of the form:
  ;;;
  ;;; (TestFn SUBSYSTEM TEST_ACTION POSSIBLE_OBSERVABLE)
  ;;;
  ;;; Result   : Prints to the screen the current 'test' parameters, i.e,
  ;;;            the SubSystem, TestAction and PossibleObservable.
  ;;; Remarks  :

  (define present-test-parameters (test)
```

```
 (format t "~%NEW TEST: ~%SubSystem: ~S ~%TestAction: ~S
     ~%Observable: ~S ~%" (second test) (third test) (fourth test))
 (csetf *terms* (cons (cons 'NEW (cons 'TEST test)) *terms*))
 )

;;; ********************************************
;;; ******** SubL CODE FOR PRESENT-TEST-RESULTS **
;;; ********************************************

;;; Function : PRESENT-TEST-RESULTS
;;; Arguments: A list of the form:
;;;
;;; (
;;;  ((?VAL . RESULT_1) (?TYPE . TYPE_1))
;;;      ...
;;;  ((?VAL . RESULT_n) (?TYPE . TYPE_n))
;;; )
;;;
;;; Result   : An enumarated menu of all possible results of the current 'test'
;;; Remarks  :

(define present-test-results (possible-results)

;; Get possible results
 (csetq results (mapcar #'get-ask-binding
                        possible-results
                        (position-list (length possible-results) 1)))

;; Get possible results' types (Not needed for the moment. The result
;; type is retreived by the 'sd-compare' function).

; (csetq result_types (mapcar #'get-ask-binding
;                               possible-results
;                               (position-list (length possible-results) 2)))

 (csetq counter 1)
 (cdolist (result results 't)
   (format t "~A. ~S~%" counter result)
   (csetf *terms* (cons (cons counter result) *terms*))
   (csetq counter (+ counter 1))
 )
 (csetf *no_of_choices* (- counter 1)) ;needed by 'menu'
 (csetf *results* results) ;needed by 'menu'
; (csetf *result_type* result_types) ; needed by 'menu'
)

;;; ********************************************
;;; ******** SubL CODE FOR POSITION-LIST *
;;; ********************************************

;;; Function : POSITION-LIST
;;; Arguments: 1. A number, n, indicating the number of bindings (lists of
;;;              doted pairs), in an ask-result of an 'fi-ask' function.
;;;            2. A number, k (1 =< k =< n), indicating which BINDING must
;;;              be returned by the 'get-ask-binding' function.
;;; Result   : A list of n elements equal to k.
;;; Remarks  : An auxiliary function. Creates the second arcument to be used
;;;            in a 'mapping' function which collects a list of BINDINGS for
;;;            the same ask variable.

(define position-list (n k)
 (csetq res ())
 (cdotimes (c n res)
  (csetq res (cons k res))
 )
 res
```

```
)

;;; **********************************************
;;; ******** SubL CODE FOR GET-ASK-BINDING ***
;;; **********************************************

;;; Function : GET-ASK-BINDING
;;; Arguments: 1. A list of dotted pairs of the form:
;;;
;;; ((?VAR1 . BINDING1)
;;;  (?VAR2 . BINDING2)
;;;  ...
;;;  (?VARn . BINDINGn)
;;; )
;;;            2. A number defining which BINDING must be returned.

;;; Result   : POSSIBLE_OBSERVABLE_VALUE
;;; Remarks  :

(define get-ask-binding (bindings_list bind_no)
 (rest (nth (- bind_no 1) bindings_list))
)


;;; **********************************************
;;; ******** SubL CODE FOR MENU ****************
;;; **********************************************

;;; Function : MENU
;;; Arguments: 1. A number from the menu of the possible results (local)
;;;            2. The list of possible results (global variable *results*)
;;;            3. The number of choices (global variable *no_of_choices*)
;;; Result   : Asserts into the KB a 'resultOfTest' assertion
;;; Remarks  :

(define menu (selection)
 (pcond
  ((cor (< selection 1) (> selection *no_of_choices*))
    (format t "~%~%You must give as an argument, a number between 1-~A"
     *no_of_choices*))
  (t (csetq test (fi-ask '(#$possibleTest ?TEST) *defaultMt*))
     (csetq test (get-ask-binding (first test) 1))
;     (csetf *result_type* (nth (- selection 1) *result_type*))
     (safe-fi :assert
      (list '#$resultOfTest test  (nth (- selection 1) *results*)) *defaultMt*))
 )
 (systematic *system*)
)



;;; The 'decompose' inference in KADS Systematic Diagnosis PSM takes as input
;;; the current #$SubSystem (#$hypothesis SUBSYSTEM) and decomposes it
;;; into its  subsystems (PART-OF-PREDICATE SUBSYSTEM PART),
;;; generating new hypotheses (#$possibleHypotheses PART).


;;; **********************************************
;;; ******** SubL CODE FOR  SD-DECOMPOSE *****
;;; **********************************************

;;; Function : SD-DECOMPOSE
;;; Arguments: None.
;;; Result   :
```

```
;;; Remarks  :

(define sd-decompose ()
 ;; Before un-asserting the current hypothesis, its functional parts
 ;;  must be saved
 (csetq ask-result (fi-ask '(#$possibleHypotheses ?H) *defaultMt*))
 (csetq in_hypotheses
  (mapcar #'get-ask-binding ask-result (position-list (length ask-result) 1)))

 ;; Before un-asserting the current hypothesis, the order of its subsystems
 ;; diagnosis must be saved
 (csetq first_to_test (fi-ask '(#$testFirst ?SYS) *defaultMt*))
 (csetq first_to_test (get-ask-binding (first first_to_test) 1))
 (csetq afters (fi-ask '(#$testAfter ?S1 ?S2) *defaultMt*))
 (csetq s1_list
        (mapcar #'get-ask-binding afters (position-list (length afters) 1)))
 (csetq s2_list
        (mapcar #'get-ask-binding afters (position-list (length afters) 2)))

;; Store hypothesis and un-assert it
 (csetq hypothesis (fi-ask '(#$hypothesis ?H) *defaultMt*))
 (csetq hypothesis (get-ask-binding (first hypothesis) 1))
 (fi-unassert (list '#$hypothesis hypothesis) *defaultMt*)

;; Assert possibleHypotheses.
 (cdolist (system in_hypotheses t)
  (safe-fi :assert (list '#$possibleHypotheses system) *defaultMt*)
 )

;; Assert diagnosis order
; (safe-fi :assert (list '#$testFirst first_to_test) *defaultMt*) ;unnecessary
 (cdolist (s1 s1_list t)
  (csetq s2 (first s2_list))
  (csetq s2_list (rest s2_list))
  (safe-fi :assert (list '#$testAfter s1 s2) *defaultMt*)
 )

;; Un-assert the (#$plausibleInference #$Decompose) assertion.
 (fi-unassert '(#$plausibleInference #$Decompose) *defaultMt*)

;; Assert the new hypothesis
 (safe-fi :assert (list '#$hypothesis first_to_test) *defaultMt*)
)

;;; *********************************************
;;; ******** SubL CODE FOR  SD-RESET ****************
;;; *********************************************

;;; Function : SD-RESET
;;; Arguments: The system that is going to be diagnosed. For the
;;;           moment, any one of the symbols '#$PCSystem' or
;;;           '#$AutomobileSystem'.
;;; Result   : Resets all assertions, in the appropriate Microtheory,
;;; regarding the following predicates :

;;;     '#$diagnosisContext
;;;     '#$hypothesis',
;;;     '#$possibleHypotheses'
;;;     '#$resultOfTest'
;;;     '#$testFirst'
;;;     '#$testAfter'
;;;     '#$diagnosisContext' (dependant from #$resultOfTest)
;;;     '#$possibleTest'     (dependant from #$hypothesis & #$resultOfTest)

;;; Remarks  :
```

```
(define sd-reset (system)

 (pcond
  ((eql system '#$PCSystem) (csetf *defaultMt* '#$PCDiagnosisMt))
  ((eql system '#$AutomobileSystem)
                    (csetf *defaultMt* '#$AutomobileDiagnosisMt))
  (t (error "SD-RESET: argument must be '#$PCSystem' or '#$AutomobileSystem"))
 ) ;end of pcond

 ;; Set the global variables
 (csetf *use-local-queue?* NIL)
 (csetf *test* nil)
 (csetf *results* nil)
 (csetf *no_of_choices* 0)
 (csetf *system* nil)

 ;;UN-ASSERT #$diagnosisContext
 (csetq diagnosisContext (fi-ask '(#$diagnosisContext ?C) *defaultMt*))
 (csetq diagnosisContext (get-ask-binding (first diagnosisContext) 1))
 (fi-unassert (list '#$diagnosisContext diagnosisContext) *defaultMt*)

 ;; UN-ASSERT #$hypothesis
 (csetq hypothesis (fi-ask '(#$hypothesis ?H) *defaultMt*))
 (csetq hypothesis (get-ask-binding (first hypothesis) 1))
 (fi-unassert (list '#$hypothesis hypothesis) *defaultMt*)

 ;; UN-ASSERT #$possibleHypotheses
 (csetq ask-result (fi-ask '(#$possibleHypotheses ?H) *defaultMt*))
 (csetq results
  (mapcar #'get-ask-binding ask-result (position-list (length ask-result) 1)))
 (cdolist (result results t)
  (fi-unassert (list '#$possibleHypotheses result) *defaultMt*)
 )

 ;; UN-ASSERT #$resultOfTest
 (csetq ask-result (fi-ask '(#$resultOfTest ?T ?R) *defaultMt*))
 (cdolist (bindings ask-result t)
  (fi-unassert
    (list '#$resultOfTest (get-ask-binding bindings 1)
                          (get-ask-binding bindings 2)) *defaultMt*)
 )

 ;;UN-ASSERT #$testFirst
 (csetq first_to_test (fi-ask '(#$testFirst ?SYS) *defaultMt*))
 (csetq first_to_test (get-ask-binding (first first_to_test) 1))
 (fi-unassert (list '#$testFirst first_to_test) *defaultMt*)

 (csetq afters (fi-ask '(#$testAfter ?S1 ?S2) *defaultMt*))
 (csetq s1_list
        (mapcar #'get-ask-binding afters (position-list (length afters) 1)))
 (csetq s2_list
        (mapcar #'get-ask-binding afters (position-list (length afters) 2)))
 (cdo
  ((s1 (first s1_list) (first s1_list))
   (s2 (first s2_list) (first s2_list))
   (s1_list (rest s1_list) (rest s1_list))
   (s2_list (rest s2_list) (rest s2_list))
  ) ;end of variables
  ((null s1)) ;when no more couples of s1,s2
  (fi-unassert (list '#$testAfter s1 s2) *defaultMt*)
 )
)
```