# Expert System Maintenance

Tim Menzies[1], John Debenham[2]

[1]Artificial Intelligence Department,
School of Computer Science and Engineering,
University of NSW, Australia, 2052
[2]School of Computer Sciences
University of Technology, Broadway, Sydney, Australia
tim@menzies.com
debenham@socs.uts.edu.au
http://www.cse.unsw.edu.au/~timm

July 21, 1998

**Abstract**

Knowledge maintenance (KM) is the process of reflecting over some current knowledge based system in order to handle a new situation. Changes that fix new problems must not introduce bugs into old fixes. A common technique for KM is to reflect over dependencies between concepts. In this review, we sample the KM work on modeling dependencies using either a logic-based, a network-based, or a procedural-based approach.

*Submitted to the Encyclopedia of Computer Science and Technology*

## 1 Introduction

A truism of software is that any working system will need to be changed. Managing change can be an expensive business. If done poorly, systems that start out as understandable and useful can quickly become incomprehensible and useless.

Maintenance is the science of controlling change in a software system. Knowledge maintenance (KM) is the science of controlling change in a knowledge base system (KBS). More precisely:

> *Knowledge maintenance (KM) is the process of reflecting over some knowledge based system in order to handle a new situation.*

An important detail in KM is that changes should improve the knowledge, not complicate or confuse it. In practice, this means that new fixes should not introduce bugs into old fixes. The article will explore how we can represent the connections between knowledge. Such connections enable effective KM that avoids breaking old fixes. However, rather than review the entire state-of-the-art in KM, this article will focus on an important common theme in many KM systems: dependency graph analysis. We will show that such an analysis is used in may different KM approaches, including:

- Logical approaches that use an abstract description of the fundamental logic of the domain. Our exemplar logical approach for maintenance will be Debenham's i-schemas and o-schemas [22].
- Procedural approaches that control the order in which items are updated. Our exemplar procedural maintenance approach will be Tallis and Gil's ETM system [30, 51].
- Network approaches that use an explicit graph showing the connections between concepts. Our exemplar network approach for maintenance will be Menzies' graph-based abductive system [40, 41].

We shall also see that the distinguishing features between different approaches are:

- How the graph is generated. Logical approaches strive to construct *clean* dependency graphs (by *clean*, we mean *normalised*: see below).
- How the graph is used. Procedural approaches strive to patch many parts of an *unclean* dependency graph. Network approaches take a dependency graph and explore what maintenance processes are supported by that graph.

The rest of this article is structured around these observations. We begin by studying how exemplars of logical, network-based, and procedural KM use a dependency graph. Next, a sample of KM systems will be described. In that sample, we shall see that much of KM can be expressed in terms of studying a dependency graph. The final discussion section will argue that dependency graph knowledge is an essential pre-condition for any maintainable language.

This article is intended as an introduction to a complex field. Hence, it is not a comprehensive review of KM. Such reviews are very large and complicated (e.g. [38]). Here, we constrain our scope. For example, one omission from this review will be systems that try to tame maintenance via the use of high-level languages (e.g. [17]). Those systems assume that if we can express an idea cleanly and succinctly enough in the first place, that subsequent change will be minimised and simplified. This review is restricted to systems which assume that change is an on-going process.

## 2 Exemplars of KM

This section studies three exemplars of KM using either a logical, procedural, or network-based approach.

### 2.1 A Logical Maintenance Tool

Our survey of exemplar approaches begins with Debenham's i-schema and o-schema logic-based KM approach [18–24].

There are at least two approaches to KM:

- Minimise the time spent on design and then try to *control* the maintenance process. Two examples of this control approach are ETM and ripple-down-rules (see below).
- *Engineer* a model so that is is in a form that is inherently easy to maintain. Three examples of the engineering approach are database normalisation [15], network-based tools (see below) and this schema-based approach.

| *structure* | |
|---|---|
| item-name | |
| var-i | var-j |
| x | y |
| meaning of item | |
| constraints on values | |
| set constraints | |

| *e.g. part* |
|---|
| part |
| - |
| x |
| isa(x:part-number) |
| $1000 < x < 9999$ |
| $\leq 100$ |

| *e.g.part/cost-price* | |
|---|---|
| part/cost-price | |
| part | cost-price |
| x | y |
| - | |
| cost(x,y) | |
| $x < 1999$ if $y \leq 300$ | |
| $\forall$ | |
| ————- | o |

Table 1: Items in i-schemas.

In logic-based approaches, finding the underlying logic is an early goal of the system's development. A software engineering process first generates a requirements model describing *what* the system should do. Next, a conceptual model is built using the schemas. Subsequent engineering that conceptual model into a physical implementation (*how* the system implements). For example, in the schema approach, the conceptual model contains *items* connected by *objects* to describe *data*, *information*, and *knowledge*:

- Items are described in i-schemas.
- Objects are described by o-schemas.
- Data are simple variables.
- Information are relations connecting variables.
- Knowledge are the rules that execute over the relations.

After the requirements modeling, the conceptual model is built using the schemas and a *coupling relationship* which lets us construct the dependency graph from the schemas. Two subsequent models are the external model (how to apply the conceptual model) followed by the implementation model (the running system). The requirements, external and implementation models will not be discussed further here.

Schemas have a uniform format, no matter if they are representing data, information, or knowledge. They offer constraints and (as we shall see below) normalisation rules that are analogous to database normalisation rules. Schemas can be represented formally using the lambda calculus or informally in a simple tabular format. Schemas can contain other schema's recursively. Three i-schemas are shown in the i-schemas of Table 1. The general structure is shown on the left. A simple example is shown in the middle. There can be no more than 100 part-numbers numbered between 1000 to 9999. A recursive shown on the right. The recursive example uses the middle example within its definition. Parts numbered under 2000 cost no more than 300 dollars. All members of the set parts must be in the set of part/cost-price (see the `forall` symbol). Horizontal lines identify components whose values determine the component marked with an `o`. For example, the last line of `part/cost-price` denotes that `cost-price` is functionally dependent on `part` (in traditional database jargon, `part` is a candidate key).

When two pieces of knowledge share the same basic wisdom, i-schemas require the wisdom be represented twice. For example:

- Suppose `part/cost-price` is computed from a function COSTS.

| *structure* | |
|---|---|
| object-name | |
| type-i | type-j |
| x | y |
| meaning of object | |
| constraints on object values | |
| object set constraints | |

| *e.g. costs* | |
|---|---|
| costs | |
| $D^1$ | $D^1$ |
| x | y |
| costs(x,y) | |
| $x < 1999$ if $y \leq 300$ | |
| $\forall$ | |
| ————- | o |

| *e.g. mark-up-rule* | | |
|---|---|---|
| part/cost-price | | |
| $I^2$ | $I^2$ | $D^1$ |
| (x,w) | (x,y) | (z) |
| $(w = z * y)$ | | |
| $w > y$ | | |
| $\forall$ | $\forall$ | |
| ————- | | o |
| o | ————- | |
| ————- | o | ————- |

Table 2: Objects in o-schemas.

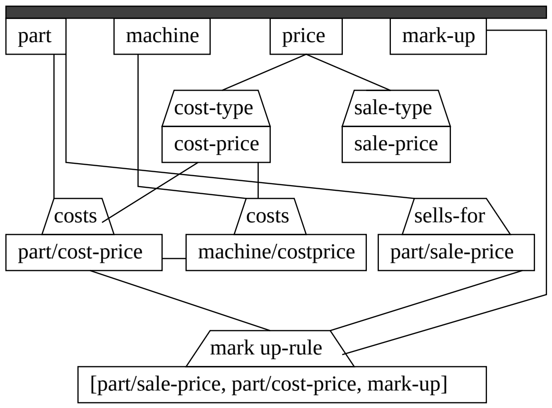

Figure 1: A simple concept map showing dependancies between schema concepts.
.

- Suppose also that another function MARK-UP-RULE uses `part/cost-price`.

Rather than represent COSTS twice (once in `part/cost-price` and once in `part/cost-price`), we can use the O-schemas shown in Table 2.

A *coupling map* shows when schemas share some common structure. For example, the compiling map for `mark-up rule` is shown in Figure 1. This map can be used to control KM. If two items are linked in the map, then modifications to one implies that the other has to be checked again for correctness. Further, these links can be used to simplify maintenance:

- An item is said to be *decomposable* if it may be constructed from other items or objects.
- If all decomposable data, information, and knowledge is discarded, then the knowledge base is said to be *normalised* [18].
- A normalised system is far simpler to maintain than an un-normalised system (a result first reported in the database community [15]). In a normalised system, knowledge used in many places is stored only once. The time required to change knowledge is reduced since that knowledge is uniquely represented in a single place in the system.

The above description is only a brief overview of the i-schema and o-schema approach to KM (for full details, see [22]). Nevertheless, the core intuition is clear: maintenance

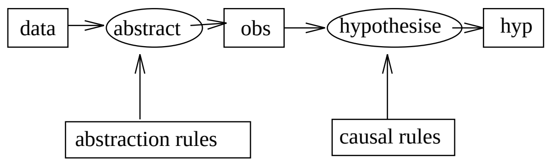

Figure 2: A diagnosis PSM.

can be simplified by converting the conceptual model into a dependency graph (which Debenham calls the coupling map).

## 2.2 A Procedural Maintenance Tool

This section describes another exemplar maintenance system: the ETM system of Gil and Tallis [30]. A first glance, the ETM will seem very different to (e.g.) the logic-based KM work described above. For example, ETM use a procedural approach to handling knowledge change. However, we shall see that ETM are based on the same core intuition as i-schemas and o-schemas.

ETM assume that knowledge is organised as a set of procedures. This is a common assumption in modern KA. Inspired by Newell's *knowledge level* approach [44, 45], many KBS use procedures assessing operators which, if applied, may take the system towards some desired goal. Collections of these operators are called problem solving methods (PSMs). In recent years, extensive libraries of PSMs have been developed and thoroughly documented for many tasks such as diagnosis, qualitative reasoning, verification, and classification (e.g. [1, 4, 6, 43, 52]). For example, Clancey [7] argues that lurking within the rules of MYCIN [5] are several PSMs (which he calls model construction operators) such as:

```
METHOD findOut (
          Uses:   subsumes/2
          Method: If an hypothesis is subsumed by other
                  findings which are not present in this
                  case then that hypothesis is wrong.
  )
```

To use procedure in a knowledge base, the knowledge engineer must supply the *subsumes/2* details, e.g.

```
subsumes(surgery, neurosurgery).
subsumes(neurosurgery, recentNeurosurgery).
subsumes(recentNeurosurgery, ventricularUrethralShunt).
```

The standard architecture for PSM-based KA is at least a two-layered system. In the bottom layer are domain-dependent facts in the language of the users. In the second layer are the PSMs, which are written in a more general language. An example PSM for diagnosis [53] is shown in shown in Figure 2. Some mapping function is defined to connect the domain-dependent language (e.g. *patient history*) to a more general, domain-independent language (e.g. *data*). Domain-specific knowledge can be used in different contexts in different ways by defining different terminological mappings between domain-dependent terms and the PSMs.

Gil and Tallis' work began with the observation that maintaining a PSM-based KB implies changing numerous procedures around a PSM. Consider the example presented in [51]: we need to change a KBS devoted to transport planning. Suppose that this system calculates duration of trips involving only ships, and that now it has to be extended to consider aircraft as well. This modification requires several individual changes. First, existing knowledge may need to be modified:

- The description of vehicles used in trips has to be extended to include aircraft in addition to ships.
- The procedures to estimate duration of the trip must be change to take into account aircraft.

Next, new knowledge may be required:

- New methods to calculate the round trip time for aircraft may be required.

Any new knowledge must be integrated into the old knowledge. For example:

- The distance traveled is used in the new method for the round trip time for aircraft and in the already existing calculation for the round trip of ships. Hence, we need to make sure that they use consistent estimates of the distance.

Note that if all these changes are not completed properly, then the system will be left in some inappropriate *incoherent* state. The database community has discussed multiple updates and incoherency for many years. *Transaction management* is a database technique for avoiding incoherency. A transaction is a set of updates that must all be completed correctly. A transaction control system manages these updates and ensures that either all the updates complete or, if any update fails, then all the completed updates are reversed. Gil and Tallis built a transaction management system for multiple procedural updates. Their *EXPECT Transaction Management* tool (or ETM) are triggered each time a user performs some initial changes to a KBS:

- The KBS is assumed to be coherent before the changes. Any incoherences subsequently detected are hence blamed on the user's recent changes.
- After the changes, automatic tools look for incoherences. ETM can recognise incoherences via a type-checking analysis of procedure input variables. More precisely, after a partial evaluation of the KBS, incoherences are detected if a method cannot fire because the types of the input parameters to the methods are not available.
- ETM examines the inconsistency and proposes a set of scripts that might fix it. These proposed scripts are taken from a hand-built library created via an analysis of common errors made by prior users of that KBS.
- Users can then either select a script to run or can fix the problem manually using standard KBS editing tools.

In one study of maintenance times by four subjects (S1..S4) and two change tasks for a KBS, maintenance was easier with ETM. The Gill & Tallis results are shown in Table 3. Note the speed up in maintenance times for two change tasks for a KBS, with and without ETM. Note also in the last row that ETM performed some automatic changes.

At first glance, ETM appears totally different to Debenham's schema approach:

|  | Simple task #1 |  |  |  | Harder task #2 |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  | no ETM |  | with ETM |  | no ETM |  | with ETM |  |
|  | S4 | S1 | S2 | S3 | S2 | S 3 | S1 | S2 |
| Total time (min) | 25 | 22 | 19 | 15 | 74 | 53 | 40 | 41 |
| Time completing transactions | 16 | 11 | 9 | 9 | 53 | 32 | 17 | 20 |
| Total changes | 3 | 3 | 3 | 3 | 7 | 8 | 10 | 9 |
| Changes made automatically | n/a | n/a | 2 | 2 | n/a | n/a | 7 | 8 |

Table 3: Change times for ETM with four subjects: S1...S4. From [30]

- Schemas are engineered to avoid the multiple update problem. ETM assumes multiple updates and tries to manage them.
- Schemas assume a logical foundation for knowledge. ETM assumes a procedural foundation for knowledge.

Surprisingly, despite these apparent differences, schemas and ETM's share the same core intuition: KM requires access to the the dependencies between concepts. Recall that incoherences were auto-detected in ETM using a type-checking system that looks for unreachable types in procedure calls. Conceptually, such a type checker builds a graph whose:

- Vertices are system types or procedures
- Whose edges are connections between types enabled by the procedures; e.g. some procedure might input *type1* and output *type2*, thus generating an edge between those types.

Such a type graph would appear very similar to the schema coupling maps.

Having described the essential commonality between ETM and schemas, we should also describe their essential difference:

- Schemas are a engineering-based KM technique that converts conceptual models to coherent and minimal (i.e. decomposable items discarded) dependency graphs.
- ETM is a control-base KM strategy that inputs potentially incoherent dependency graphs (the type graph) and outputs a KBS whose dependency graph may contain less incoherences.

## 2.3 A Network Maintenance Tool

Schema-based KM defines only one operation on KBS dependency graphs (which Debenham calls coupling maps): the removal of decomposable items. ETM-based KM defines a library of hand-crafted dependency graph operations: these ETM operations take the form *if problem X then perform updates Y*. Menzies' network-based KM tries to generalise dependency graph processing. His HT4 system [39, 41] supports an extensive library of dependency graph operations. This section describes those operations.

Both schema-based KM and ETM-based KM make strong assumptions about the representations they process:

- Schema-based KM is defined only for i-schema and o-schemas.

```
rule1 if happy
      then motivated.
rule2 if happy and well-feed
      then lazy.
rule3 if motivated
      then not lazy.
rule4 if not lazy
      then vigilant.
```

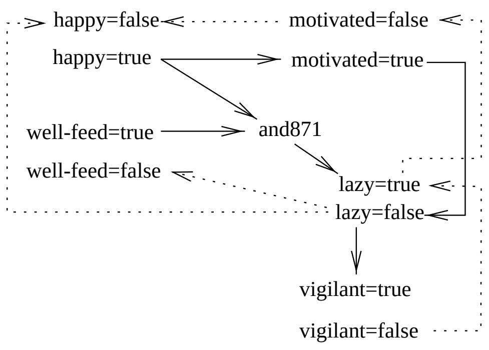

Figure 3: An example of generating `T` (right) from some rules (left). Modus tollens links shown as dashed lines.

- ETM is tightly integrated into the EXPECT KA tool which is a LOOM-based representations (a specific-type of object-oriented expert system).

In contrast, HT4 assumes that some other program has taken some domain-specific representation and converted that into a dependency graph. HT4's maintenance tools are defined for any theory that can be converted to the form `T=(V,Ed,I)` where:

- Each vertex `V.i` represents the assignment of a value to a variable. `V.i` can be an AND-node or an OR-node. If an AND-node appears in a proof, then all its parents must also appear in that proof. If an OR-node appears in a proof, then one of its parents must also appear in that proof.

- Each directed edge `Ed.i` represents a statement of the possibility that the variable-value assignment in the originating node may lead to the variable-value assignment in the terminating node. Each edge `Ed.i` is tagged with the author(s) who proposed it.

- An integrity constraint `I` complains if an illegal combination of value assignments are being made to variables. For example, `I` could complain if we tried to assign both *up* an *down* to the direction of change of a variable.

- `T` is some theory generated from the user's assertion about their domain. Examples of this generation process are shown in Figure 3 and in Figure 4.

Theories in the form of `T` offer many maintenance operations such as verification, validation, and supporting arguments. Many tests of a model can also be characterised as dependency graph traversal. Testing is often divided into verification (was the system built right?) and validation (was the right system built?). A verification system can detect (e.g.) loops, unreachable conclusions and tautologies [47]. Note that these verification tools can be viewed as dependency path traversal analysis. For example:

- Report a loop if a path comes back to itself.

- Report a tautologies if two paths reach the same conclusion.

- Report an unreachable conclusion if no path can find that conclusion.

A validation system can use a test suite to check if paths can be found from test inputs to test outputs. If a model does not contain consistent paths from known inputs to known

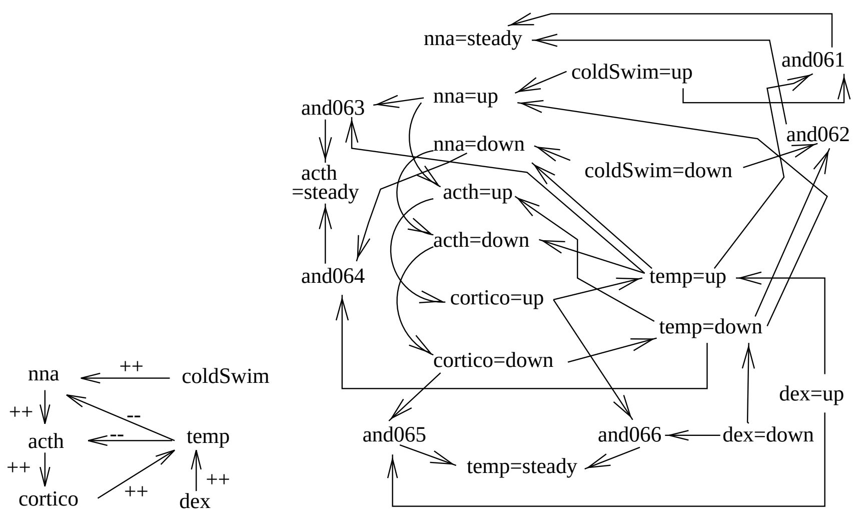

Figure 4: The Smythe'87 [50] model of stress regulation (left) and its associated theory `T` (right). In the language of the Smythe'87 theory, variables have three states: `up`, `down` or `steady`. Competing influences can cancel out to explain a steady. For example, note that there are two upstream influences to *nna*. Therefore, *nna* being steady can be explained by a conjunction of (e.g.) *coldSwim=up* and *temp=down*.

outputs, then that model is faulty. Feldman and Compton [26], followed by Menzies [39, 41], have shown that this test procedure can detect a large number of previously unseen errors in models taken from international refereed scientific publications. Even when they searched for consistent pathways that could explain as many observations as possible, many observations were still inexplicable. The detected faults were surprising and insightful to the authors of the publications.

The validation technique used Feldman, Compton, and Menzies is illustrated using an economics model written in the QCM language [41]. The model is shown in Figure 5. In QCM , theory variables have three values: *up*, *down* or *steady*. The direct connection between *foreignSales* and *companyProfits* (denoted with plus signs) means that *companyProfits* being *up* or *down* should be connected back to *foreignSales* being *up* or *down* respectively. The inverse connection between *publicConfidence* and *inflation*

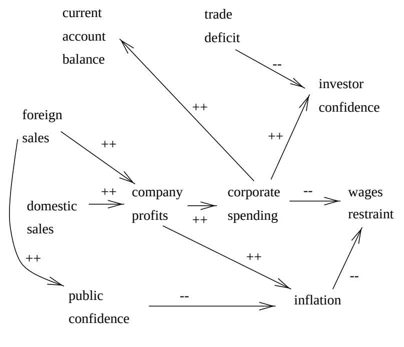

Figure 5: An economics model.

```
P.1   foreignSales=up, companyProfits=up, corporateSpending=up,
      investorConfidence=up.
P.2   domesticSales=down, companyProfits=down,
      corporateSpending=down wageRestraint=up.
P.3   domesticSales=down, companyProfits=down, inflation=down.
P.4   domesticSales=down, companyProfits=down, inflation=down,
      wagesRestraint=up.
P.5   foreignSales=up, publicConfidence=up, inflation=down.
P.6   foreignSales=up, publicConfidence=up, inflation=down,
      wageRestraint=up.
```

Table 4: Pathways that explain outputs in the economics model.

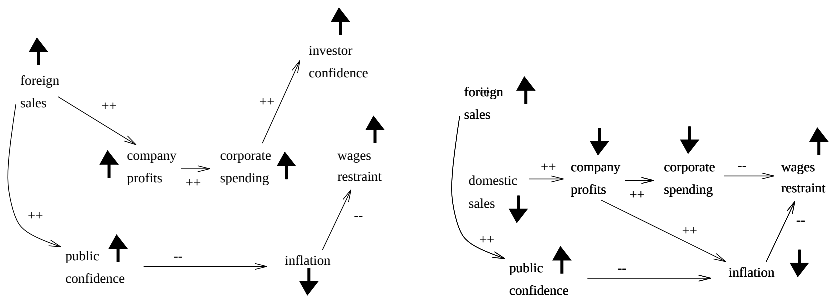

Figure 6: Explanation `E.1`, left; Explanation `E.2`, right.

(denoted with minus signs) means that *inflation* being *up* or *down* should be connected back to *publicConfidence* being *down* or *up* respectively. In the case where the inputs are *(foreignSales=up, domesticSales=down)* and the output goals are *(investorConfidence=up, inflation=down, wageRestraint=up)*, then certain some pathways can be generated to explain the outputs. Note that some of these pathways make contradictory assumptions; e.g. e.g. *corporateSpending=up* in `P.1` and *corporateSpending=down* in `P.2`. That is, we cannot believe in `P.1` and `P.2` at the same time. If we sort these pathways into the biggest possible sets that can be believed at the same time, we arrive at the two consistent sets of explanations shown in Figure 6. Explanation `E.1` contains 3 pathways that can be believed at the same time; i.e. (`P.1, P.5, P.6`) while explanation `E.2` contains 4 pathways that can be believed at the same time; i.e. (`P.2, P.3, P.4, P.6`). Note that our explanations can explain different output goals. `E.1` can explain all the goals while `E.2` can only explain two of the three output goals. Formally, this extraction and assessment of consistent inferences from a dependency network is graph-based abduction [41]. Feldman, Compton, and Menzies faulted theories when those theories could not generate explanations for all known output goals.

Arguments between developers are very common during maintenance. Graph-based abduction offers numerous support tools for argumentation. Recall that each edge `Ed.i` in `T` is tagged with the name(s) of its author. True conflicts between users can be detected when their edges end up in different explanations. False conflicts can be quickly resolved if, in no real-world example, the views of different users end up in different explanations. True conflicts can then be negotiated by focusing on the key conflicting assumptions that split the paths (see the discussion below on fault localisation). If negotiation fails, disputes between feuding users can be adjudicated using model assessment. The winning user offers theories which build consistent paths that can cover most of the desired behaviour.

When shared perspectives are being built and maintained collaborative by groups, the system must offer awareness management tools [2]; i.e. monitoring that someone else is not making changes that unduly influence your contribution to the shared perspective. If models perspectives are expressed in T format, users could be alerted if:

- Some changes mean that the pathways they have proposed have been removed, and...
- The system's performance is degrading as a result of those changes.

HT4's graph-based abduction can be compared with schemas and ETM as follows:

- All these techniques (based on different logics, procedures, or networks) utilise a dependency graph.
- I-schemas and O-schemas is a method for building *clean* (i.e. normalised) dependency graphs.
- ETM is a method for building macros that fix common errors in bad dependency graphs. These fixes can imply patches in multiple places around the system that generated the dependency graph.
- HT4 is a method for processing dependency graphs, once they are created. Note that this processing can handle inconsistent dependency graphs: such graphs generate multiple explanations.

## 3 Other KM Systems

This section reviews a range of KM systems that allow a user to:

- *Browse-around*: explore the connections between knowledge inside their KBS.
- *Fault localisation*: find the reason why certain undesirable behaviour has occoured.
- *Fix*: change the KBS to correct undesirable behaviour.
- *Annotate*: add a memo to explain why the change was made.

A key data structure used by all these systems will be the dependency graph between concepts in the KBS.

### 3.1 Browse-Around

Browse-around tools let a user manually generate their own explanations of why a variable was/was not set. Browse-around is implemented via queries to the dependency graph from a KB. The user starts at some point in the KB and explores the nearby region. This section describes sample browse-around tools are described below.

The MYCIN rule-based system [5] maintained connection knowledge between fired rules. MYCIN's *how* and *why* queries allowed the user to start at a conclusion or a question and ask *how was that conclusion reached?* or *why are you asking me* *that question?*. Both queries reported the dependency links in the neighborhood of the conclusion or question. How-queries looked upstream back towards the other literals that lead to this literal. Why-queries looked downstream to the word that is the current goal of the system. (Technical note: in backward chaining systems like MYCIN, there is always a goal that the system is trying to prove.)

SALT [36] was a general KA shell to support the construction of propose and revise design systems. The SALT environment was a generalisation of a suite of tools built for the VT elevator configuration system [37]. SALT supported how-queries in the MYCIN-sense. However, SALT why-queries just returned canned text since why-queries have a natural meaning in goal-directed systems like MYCIN. SALT was a forward-chainer: it's rules fired according to the supplied data. The goal of firing a particular forward chaining rule is hence not as clear cut as firing a backward chaining rule. SALT also supported *why not* and *what if* browse around tools. Answers to *why not value X?* were generated by seeking other words that could lead to *X*, but which were blocked somehow by known words. The system could answer (e.g.) *if Z=1 was set to Z=2, then we could have achieved X*. A what-if-query was a hypothetical look downstream of some temporary setting of a variable.

Fensel and Schoenegge [27] offer an interesting variant on *browse-around*. Using an interactive theorem prover (KIV), they let a user browse-around a first-order theory representing a PSMs. If KIV cannot solve a problem, it identifies the missing logical formula that blocked PSM completion and presents this to the user as an assumption to be explored. Users of KIV can hence discover what extra assumptions are required to achieve their desired goals.

## 3.2 Fault Localisation

A general technique for finding a fault in a system is dependency tracing. If the dependencies within a KB are known, and the inferencing has arrived in some unexpected part of that dependency graph, then fault localisation can be implemented via a backwards search of the dependencies. All the above browse-around tools can be described as manual fault localisation systems. This section describes a range of automatic or semi-automatic fault localisation schemes.

### 3.2.1 Test Suites: Creation and Usage

To find a fault, a system must be tested. To ensure that a system has no faults, all parts of the dependency network must be exercised by the test suite. Proposed test suites can be assessed by what percentage of the paths in a model were exercised during the execution of that test suite [12]. Alternatively, such test suites can be automatically generated by looking for tests that cover all branches in the paths [54].

### 3.2.2 Theory Evolution

Darden [14] offers an account of the evolution of gene theory in the period 1900-1930. In terms of improving a theory, the key requirement of a representation was that it could generate a directed graph showing the influences between variables [13]. In the language of this paper, such a directed graph is the KBS dependency graph. The natural language of experts trying to refine system behaviour often takes the form of a traversal of the dependency graph; e.g. *Let's see now: after the inflation rate rose and* *drove interest rates down, then we saw an increase in home occupancy rates. OK, lets try increasing the interest rates.*

### 3.2.3 TEIREISIAS

The MYCIN rule editor, TEIREISIAS, applied a clustering analysis to the rule base to determine what parameters where *related*; i.e. are often mentioned together. If proposed rules referred to a parameter, but not its related parameters, then TEIREISIAS would point out a possible error [16]. Note that clustering can be simply defined using graph-based techniques.

### 3.2.4 Model-based Diagnosis

Fault localisation is a explored extensively in the model-based diagnosis literature [31]. A commonly used support-routine for diagnosis is probing; i.e. the guided search for additional information which can confirm/ rule-out a diagnosis. Given a set of possible explanations, a carefully selected probe for a single piece of information can cull numerous explanations. DeKleer uses the dependency structures of his general diagnosis engine GDE to guide probe selection [25]. The usual case is that each probes have an associated cost (e.g. taking a blood- pressure reading is a cheap probe while performing exploratory surgery is an expensive probe). Probe selection is a trade-off between information gain and probe cost. Information gain can be expressed in terms of dependency graphs. For example, in the case of HT4, an intelligent probing strategy would be to only probe on the most upstream controversial assumptions. Returning to the above HT4 example, consider the assumptions relating to *companyProfits*, *corporateSpending* and *publicConfidence* (an assumption in graph-based abduction is any value assignment used in proofs which does not appear in the inputs or outputs). Note that our explanations disagree about *companyProfits* and *corporateSpending*, but agree about *publicConfidence*. Hence, intelligent probes would ignore checking *publicConfidence* (since no one is arguing about that) or *corporateSpending* (since it is fully dependent on the upstream concept of *companyProfits*).

### 3.2.5 Ripple-Down-Rules

Ripple-down-rules (RDR) is a representation optimised for fault localisation in KBS without PSMs [8–10, 48]. RDR knowledge is organised into a *patch tree*. If a rule is found to be faulty, some patch logic is added on a *unless* link beneath the rule. The patch is itself a rule and so may be patched recursively. Whenever a new patch (rule) is added to an RDR system, the case which prompted the patch is included in the rule. These *cornerstone cases* are used below when fixing an RDR system (see below). At runtime, the final conclusion is the conclusion of the last satisfied rule. If that conclusion is faulty, then the fault is localised to the last satisfied rule. Note that the RDR patch tree is a dependency graph connecting rule logic.

## 3.3 Fix

Once a fault is localised, it must be fixed. This section describes a range of fixing schemes. Dependency graphs will be seen to be central to many of the fix strategies.

The RDR representation is useful for both fault localisation and fixing. Once an expert has faulted a conclusion from an RDR system, they then ask the system for a

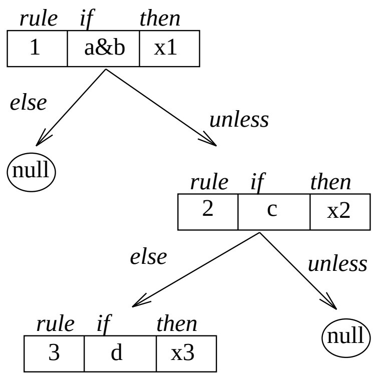

Figure 7: An RDR knowledge base

list of possible patches. The system replies with a *difference list* which is calculated as follows. As the *current case* navigates down the RDR tree, if it finds a some satisfied rule, it then checks their *unless* patches. The different between the current case and the cornerstone case of the last satisfied rule is the difference list. For example, a sample RDR KB is shown in Figure 7. The rule *if a and b then x1* has been patched several times. Suppose:

- *x2* is the correct conclusion when *a and b and c* is true.
- But later on we realise that when when *c* is false and *d* is true, then *x3* is true.

RDR would add a logic delta *not c* to a patch rule in the *unless* branch beneath *rule 2*.

In practice, RDR appears to work very well, at least for single classification problems. For example, the PIERS system at Sydney's St. Vincent's Hospital,models 20 percent of human biochemistry sufficiently well to make diagnoses that are 99 percent accurate [48]. System development blends seamlessly with system maintenance since the only activity that the RDR interface permits is patching faulty rules in the context of the last error. For a 2000-rule RDR system, maintenance is very simple (a total of a few minutes each day). RDR has succeeded in domains where previous attempts, based on much higher-level constructs, never made it out of the prototype stage (e.g. [46]). Further, while large expert systems are notoriously hard to maintain [17], the no-model approach of RDR has never encountered maintenance problems.

Other fix strategies focus on editing the dependencies upstream of an error. *Specialisation/ generalisation* is a general framework for repairing logic. The framework dates back to at least Shapiro [49]. Undesired behaviours can be removed by specialising a pre-condition; i.e. increasing the number of tests in a conjunction. Desired behaviour which was not achieved can be reached via generalising a pre-condition; i.e. decreasing the number of tests in a conjunction. In graph-theoretic terms, specialisation and generalisation amounts to removing or adding edges above AND-nodes in a dependency graph.

Representation-specific variants of this framework have appeared in various systems; for example, Shapiro's system, the work of van Harmelen and Aben, and the SEEK/SEEK2 systems. Shapiro's own system was a debugging facility for horn clauses. In that system, specialisation or generalisation means adding or removing (respectively) horn clause sub-goals. Fixing in the SEEK system used explicit fix knowledge Example SEEK rules are shown in Figure 8. SEEK rules can be specialised by adding

```
seekDiagnosisRule
if   majorSymptoms and
     minorSymptoms and
     tests and
     not exclusions
then diagnosis is true
with confidence
     (definitely or probably
      or possibly)
```

```
seekFixRule
if   the number of cases suggesting
     generalisation is greater than
     the number of cases suggesting
     specialisation
and  the most frequently missing
     component is the major symptoms
then delete some major symptoms
```

Figure 8: SEEK rules.

tests/symptoms or deleting exclusions or decreasing its confidence level. Similarly, such rules can be generalised by removing tests/symptoms or adding exclusions or increasing its confidence level. SEEK worked in association with a human operator. Its successor, SEEK was a fully automatic system.

van Harmelen and Aben [53] discuss formal methods for repairing PSMs such as the diagnosis PSM shown above. For example, that diagnosis PSM can be formally represented as a mapping from data *d* to an hypothesis *h* via intermediaries *Z* and other data *R.i*:

```
abstract(data(d),R.1,obs(Z)) and hypothesize(obs(Z),R.2,hyp(h))
```

There are three ways this process can fail:

1. We fail to prove *abstract(data(d),R.1,obs(Z))*; i.e. we are missing abstraction rules that map *d* to observations.

2. We fail to prove *hypothesize(obs(Z1),R.2,hyp(h))*; i.e. we are missing causal rules that map *Z1* to an hypothesis *h*.

3. We can prove either subgoal of our process, but not the entire conjunction; i.e. there is no overlap in the vocabulary of *Z* and *Z1* such that *Z=Z1*.

Case 1 and case 2 can be fixed by adding rules of the missing type. Case 3 can be fixed by adding rules which contain the overlap of the vocabulary of the possible *Z* values and the possible *Z'* values. More generally, given a conjunction of sub-goals representing a PSM, fixes can be proposed for any sub-goal or any variable that is used by more than one sub-goal.

## 3.4 Annotate

Fixing a KBS means changing it. Having access to the reasons for changes to the shared perspective is very important [3, 11, 29]. Studies with real-world maintainers show that the most important question a maintainer ever asks is *why did they do this, and not that?* [42]. This section discusses issues with annotating the reasons for change. Dependency-based techniques will be used to reduce the cost of generating annotations.

Argumentation structures can record prior debates [11, 32, 34, 35]. However, such argumentation structures can be very costly to build. To reduce that cost, some argumentation systems (e.g. [28]) tightly integrate the argumentation environment with the design environment:

- To check a new argument a simulation module is also offered which allows the user to make what-if queries.
- New arguments can be matched into a library of old arguments. Holes in the new argument can then be filled in automatically and offered back to the user (e.g. ‘You said this before, is this what you mean now?’).

This functionality can be implemented as graph-based abduction over dependency graphs as follows:

- Simulation models performing what-if queries is a synonym for generating multiple explanations in abduction (e.g. generating the explanations from the economics model).
- One commonly used argumentation representation [35] connects different options to assessment criteria via qualitative statements of *supports* and *objects-to*. To process this kind of argumentation, we need to build explanations containing consistent guesses about the implications of different options.
- Matching new arguments to old arguments is case-based reasoning which Leake argues is an abductive task; e.g. select the explanations containing the most number of things we have used successfully before [33].

## 4 Discussion

The observation made above was that dependency knowledge was a core data structure in many KM methods. While the use of such dependency graphs may not be immediately obvious, it can usually be found somewhere within a KM system. This discussion section offers two reflections on the ubiquity of dependency graphs for KM.

Firstly, KA researchers may not surprised at the widespread use of the dependency relationships between KBS concepts. One of the key ideas in modern KA research is Newell’s knowledge-level principle [44, 45]. At Newell’s knowledge-level, intelligence is modeled as a search for appropriate *operators* that convert some *current state* to a *goal state*. Domain-specific knowledge in used to select the operators according to *the principle of rationality*; i.e. an intelligent agent will select an operator which its knowledge tells it will lead the achievement of some of its goals. A search space is the implicit dependencies within a system (but sometimes the search space is extended at runtime when new concepts are created on-the-fly). Note Newell’s core intuition: the search space knowledge is essential to describing the runtime behaviour of an intelligent agent. This article has argued that accessing that search space, in the form of a dependency network, is a common feature of many KM schemes. Combining the views of Newell and this article, we can say that dependency graphs are a core data structure for *both* the initial acquisition and the subsequent maintenance of knowledge.

Secondly, based on the above observations, we can say that a minimal pre-condition of declaring a representation to be maintainable is:

- A dependency network can be generated, and..
- That dependency network can be extensively queried, and..
- That dependency network can be extensively updated.

This minimal pre-condition for maintainability has implications for standard software engineering. Perhaps the reason why conventional software systems are so hard to maintain is that common commercial languages such as Visual Basic and C++ do not give the developer adequate access to the dependency network. Looking into the future, we can guess that dependency network access will be a core feature of the next generation of commercial languages.

## References

[1] R. Benjamins. Problem-Solving Methods for Diagnosis and their Role in Knowledge Acquisition. *International Journal of Expert Systems: Research & Applications*, 8(2):93–120, 1995.

[2] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel. Basic Support for Cooperative Work on the World Wide Web. *International Journal of Human Computer Studies*, 46(6):827–846, June 1997. Available from `http://bscw.gmd.de/Papers/IJHCS/IJHCS.html`.

[3] G. Boy. Supportability-based design rationale. In *Proceedings of the 6th IFAC Symposium on Analysis, Design and Evaluation of Man-Machine Systems*, 1995.

[4] J. Breuker and W. Van de Velde (eds). *The CommonKADS Library for Expertise Modelling.* IOS Press, Netherlands, 1994.

[5] B.G. Buchanan and E.H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, 1984.

[6] B. Chandrasekaran, T.R. Johnson, and J. W. Smith. Task Structure Analysis for Knowledge Modeling. *Communications of the ACM*, 35(9):124–137, 1992.

[7] W.J. Clancey. Model Construction Operators. *Artificial Intelligence*, 53:1–115, 1992.

[8] P. Compton, G. Edwards, A. Srinivasan, P. Malor, P. Preston, B. Kang, and L. Lazarus. Ripple-down-rules: Turning Knowledge Acquisition into Knowledge Maintenance. *Artificial Intelligence in Medicine*, 4:47–59, 1992.

[9] P. Compton, K. Horn, J.R. Quinlan, and L. Lazarus. Maintaining an Expert System. In J.R. Quinlan, editor, *Applications of Expert Systems*, pages 366–385. Addison Wesley, 1989.

[10] P.J. Compton and R. Jansen. A Philosophical Basis for Knowledge Acquisition. *Knowledge Acquisition*, 2:241–257, 1990.

[11] J.E. Conklin and M.L. Begeman. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Office Information Systems*, 6:303–331, 1988.

[12] M. Connell and T.J. Menzies. Quality Metrics: Test Coverage Analysis for Smalltalk. In *Tools Pacific, 1996, Melbourne*, 1996.

[13] L. Darden. Diagnosing and Fixing Faults in Theories. In J. Sharager and P. Langley, editors, *Computational Models of Scientific Discovery and Theory Formation.* Morgan Kaufmann Publishers Inc., 1990.

[14] L. Darden. *Theory Change in Science: Strategies from Mendelian Genetics.* Oxford University Press, 1991.

[15] C.J. Date. *An Introduction to Database Systems*, volume 6. Addison-Wesley, 1995.

[16] R. Davis. Interactive Transfer of Expertise: Acqusiition of New Inference Rules. *Artificial Intelligence*, 12(2):121–157, 1979.

[17] A. Van de Brug, J. Bachant, and J. McDermott. The Taming of R1. *IEEE Expert*, pages 33–39, Fall 1986.

[18] J. Debenham. Understanding Expert Systems Maintenance. In *Proceedings Sixth International Conference on Database and Expert Systems Applications DEXA'95, London, September,* 1995.

[19] J. Debenham. A Unified Approach to Requirements Specification and Systems Analysis in the Design of Knowledge-Based Systems. In *Proceedings Seventh International Conference on Software Engineering and Knowledge Engineering SEKE'95, Washington, June*, 1995.

[20] J. Debenham. Knowledge Simplificiation. In *Proceedings 9th International Sympoisum on Methodologies for Intelligent Systems ISMIS '96, Zakopane, Poland, June*, pages 705–314, 1996.

[21] J. Debenham. An Integrated Conceptual Model of Knowledge-Based Systems Simplifies Mainteance. In *Proceedings Seventh International Conference on Intelligent Systems ICIS'98, Paris, France, July*, 1998.

[22] J. Debenham. *Knowledge Engineering: Unifying Knowledge Base and Database Design*. Springer-Verlag, 1998.

[23] J. Debenham. Managing Knowledge Integrity. In *Proceedings Seventh International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems IPMU '98, Paris, France, July*, 1998.

[24] J. Debenham. Representing Knowledge Normalisation. In *Proceedings Tenth International Conference on Software Engineering and Knowledge Engineering SEKE'98, San Francisco, US, June*, 1998.

[25] J. DeKleer and B.C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32:97–130, 1 1987.

[26] B. Feldman, P. Compton, and G. Smythe. Towards Hypothesis Testing: JUSTIN, Prototype System Using Justification in Context. In *Proceedings of the Joint Australian Conference on Artificial Intelligence, AI '89*, pages 319–331, 1989.

[27] D. Fensel and A. Schoenegge. Hunting for Assumptions as Developing Method for Problem-Solving Methods. In *Workshop on Problem-Solving Methods for Knowledge-based Systems, IJCAI '97, August 23.*, 1997.

[28] G. Fischer, A.C. Lemke, R. McCall, and A.I. Morch. Making Argumentation Serve Design. In T.P. Moran and J.M. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 267–293. Lawerence Erlbaum Associates, 1996.

[29] G. Fischer, R. McCall, and A. Morch. Design environments for constructive and argumentative design. In *CHI '89*, 1989.

[30] Y. Gil and M. Tallis. A Script-Based Approach to Modifying Knowledge Bases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[31] W. Hamscher, L. Console, and J. DeKleer. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[32] M. Klein. Capturing Design Rationale in Concurrent Engineering Teams. *IEEE Computer*, 26(1):39–47, 1993.

[33] D.B. Leake. Focusing Construction and Selection of Abductive Hypotheses. In *IJCAI '93*, pages 24–29, 1993.

[34] J. Lee and K. Lai. What's in Design Rationale? In T.P. Moran and J.M. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 21–52. Lawerence Erlbaum Associates, 1996.

[35] A. MacLean, R.M. Young, V. Bellotti, and T.P. Moran. Questions, options and criteria: Elements of design space analysis. In T.P. Moran and J.M. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 53–106. Lawerence Erlbaum Associates, 1996.

[36] S. Marcus and J. McDermott. SALT: A Knowledge Acquisition Language for Propose-and-Revise Systems. *Artificial Intelligence*, 39:1–37, 1 1989.

[37] S. Marcus, J. Stout, and J. McDermott. VT: An Expert Elevator Designer That Uses Knowledge-Based Backtracking. *AI Magazine*, pages 41–58, Winter 1987.

[38] T. Menzies. Knowledge Maintenance: The State of the Art. Technical report, The Knowledge Engineering Review, 1998. To appear. Available from `http://www.cse.unsw.EDU.AU/~timm/pub/docs/97kmall.ps.gz`.

[39] T.J. Menzies. *Principles for Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales. Avaliable from `http://www.cse.unsw.edu.au/~timm/pub/docs/95thesis.ps.gz`, 1995.

[40] T.J. Menzies. Applications of Abduction: Knowledge Level Modeling. *International Journal of Human Computer Studies*, 45:305–355, September, 1996. Available from `http://www.cse.unsw.edu.au/~timm/pub/docs/96abkl1.ps.gz`.

[41] T.J. Menzies and P. Compton. Applications of Abduction: Hypothesis Testing of Neuroendocrinological Qualitative Compartmental Models. *Artificial Intelligence in Medicine*, 10:145–175, 1997. Available from `http://www.cse.unsw.edu.au/~timm/pub/docs/96aim.ps.gz`.

[42] T.P. Moran and J.M. Carroll. *Design Rationale: Concepts, Techniques, and Use*. Lawerence Erlbaum Associates, 1996.

[43] E. Motta and Z. Zdrahal. Parametric Design Problem Solving. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop*, 1996.

[44] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.

[45] A. Newell. Reflections on the Knowledge Level. *Artificial Intelligence*, 59:31–38, Feburary 1993.

[46] R.S. Patil, P. Szolovitis, and W.B Schwartz. Causal Understanding of Patient Illness in Medical Diagnosis. In *IJCAI '81*, pages 893–899, 1981.

[47] A.D. Preece. Principles and Practice in Verifying Rule-based Systems. *The Knowledge Engineering Review*, 7:115–141, 2 1992.

[48] P. Preston, G. Edwards, and P. Compton. A 1600 Rule Expert System Without Knowledge Engineers. In J. Leibowitz, editor, *Second World Congress on Expert Systems*, 1993.

[49] E. Y. Shapiro. *Algorithmic program debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[50] G.A. Smythe. Hypothalamic noradrenergic activation of stress-induced adrenocorticotropin (ACTH) release: Effects of acute and chronic dexamethasone pre-treatment in the rat. *Exp. Clin. Endocrinol. (Life Sci. Adv.)*, pages 141–144, 6 1987.

[51] M. Tallis. A Script-Based Approach to Modifying Knowledge-Based Systems. In *Banff KAW '98 workshop.*, 1998. Available from `http://ksi.cpsc.ucalgary.ca/KAW/KAW98/tallis`.

[52] D.S.W. Tansley and C.C. Hayball. *Knowledge-Based Systems Analysis and Design.* Prentice-Hall, 1993.

[53] F. van Harmelen and M. Aben. Structure-Preserving Specification Languages for Knowledge-Based Systems. *International Journal of Human-Computer Studies*, 44:187–212, 1996.

[54] N. Zlatereva and A. Preece. State of the Art in Automated Validation of Knowledge-Based Systems. *Expert Systems with Applications*, 7:151–167, 2 1994.