UDC 681.5.01

An Analysis of Knowledge Base Maintenance

John Debenham

Faculty of IT, University of Technology, Sydney
PO Box 123, Broadway 2007, Australia
debenham@it.uts.edu.au

Abstract. Knowledge base maintenance is managed by constructing a formal model. In this model the representation of each chunk of knowledge encapsulates the knowledge in a set of declarative rules, each of which in turn encapsulates the knowledge in a set of imperative programs. In this model an "item" is the unit of knowledge representation. Items are at a higher level of abstraction than rules. Understanding what has to be done to maintain the integrity of an item leads to a specification of the modifications to the set of programs that implement it. An analysis of the maintenance of the formal model is achieved by introducing maintenance links. Analysis of the maintenance links shows that they are of four different types. The density of the maintenance links is reduced by transforming that set into an equivalent set. In this way the knowledge base maintenance problem is analysed and simplified. A side benefit of knowledge items as a formalism is that they contain knowledge constraints that protect the knowledge from unforeseen modification.

1. Introduction

The problem of maintaining the consistency of a first-order knowledge base is not computable. This means that no algorithmic method can address the knowledge base integrity maintenance problem [1]. But it does not mean that the maintenance problem is not worth analysing, or that an analysis of it can not lead to its simplification, or that such simplifications can not have practical value. Here the knowledge base maintenance problem is analysed in terms of four kinds of "maintenance link". Two of these kinds of link can in principle be removed completely but not with an algorithm, another may be simplified with a method, and the final kind can not be simplified or removed.

The resulting set of links admits no further simplification and so in that sense is optimal for the representation chosen.

This analysis is made possible by the choice of knowledge model. In this model knowledge is represented at a high level of abstraction. The representation of each chunk of knowledge encapsulates the knowledge in a set of declarative rules, each of which in turn encapsulates the knowledge in a set of imperative programs. So this analysis has implications for the maintenance of imperative programs as well as for knowledge bases. In this representation each chunk of knowledge is represented as an "item" [2]. An item admits one or more declarative interpretations as ifthen rules that share common wisdom. In turn, each if-then declarative interpretation admits one or more imperative interpretations as programs. In particular, if two programs are instances of the same knowledge item then it may be the case that modification of one of them means that the other too should be modified. Items are a *uniform* representation for knowledge in the sense that all "data", "information" and "knowledge" things are represented in the same way. The insights gained by analysing the maintenance of items leads naturally to an understanding of the maintenance of if-then rules, conventional programs and other knowledge representation paradigms [3]. The integrity of items represented in this model is maintained by following maintenance links — the structure of these links is simplified by a decomposition process.

For either a knowledge base implementation or an imperative implementation, the maintenance problem is to determine which programs in it should be checked for correctness in response to a change in the application [4]. Given any form of conceptual model for knowledge, *maintenance links* may be introduced that join two things in that model if a modification to one of them means that the other must necessarily be checked for correctness, and so possibly modified, if consistency of that model is to be preserved. If that other thing requires modification then the links from it to yet other things are followed, and so on until things are reached that do not require modification. If node A is linked to node B which is linked to node C then nodes A and C are *indirectly* linked. In a *coherent* model of an application everything is indirectly linked to everything else. A good conceptual model for maintenance will have a low density of maintenance links [5]. Ideally, the set of maintenance links will be *minimal* in than none may be removed. Informally, one conceptual model is "better" than another if it takes less effort to validate it. The aim of this work is to generate a good conceptual model. A classification of maintenance links into four classes is given here. Methods are given for removing two of these classes of link so reducing the density of maintenance links in the resulting model. In this way the maintenance problem is simplified.

Approaches to the maintenance of declarative conceptual models are principally of two types [6]. First, approaches that take a model 'as is' and then try to control the maintenance process [7]. Second, approaches that *engineer* a model so that it is in a form that is inherently easy to maintain [8] [9]. The approach described here is of the second type because maintenance is driven by a maintenance link structure that is simplified by transforming the model.

The majority of conceptual models for knowledge-based systems treat the "rule base" component separately from the "database" component. This enables well established design methodologies to be employed, but the use of two separate models means that the interrelationship between the things in these two models cannot be represented, integrated and manipulated naturally within the model [4]. Further, neither of these two separate models is able to address completely the validity of the whole knowledge base.

The terms data, information and knowledge are used here in the following sense. The *data* things in an application are the fundamental, indivisible things. Data things can be represented as simple constants or variables. If an association between things *cannot* be defined as a succinct, computable rule then it is an *implicit* association. Otherwise it is an *explicit* association. An *information* thing in an application is an implicit association between data things. Information things can be represented as tuples or relations. A *knowledge* thing in an application is an explicit association between information and/or data things. Knowledge can be represented either as programs in an imperative language or as rules in a declarative language.

2. Maintaining knowledge in implementable representations

A simple example is used to motivate this discussion by examining the issues with maintaining two simple chunks of 'system knowledge' as represented in both a conventional imperative formalism and in a rule-based declarative formalism. The first chunk is: [K1] "The sale price of a part is the cost price of that part marked up by the mark-up rate for that part". The second chunk is: [K2] "The profit on a part is the difference between the marked-up cost price and the raw cost price".

Even if these two chunks of knowledge are valid they may contain a potential 'maintenance hazard'. For example, if [K3] "The profit on a part is the difference between the sale price and the cost price of that part" is valid then [K2] may be derived from both [K1] and [K3]. Why does this present a

problem? If the knowledge in either [K1] or [K3] becomes invalid and so is modified then [K2] should be modified as well. The relationship between these three chunks is not difficult to identify given the raw chunks, but, as is illustrated below, given only implementations of particular interpretations of those chunks — even with documentation — the relationship becomes noticeably more obscure.

2.1. Imperative representations

Chunk [K1] leads to a number of different imperative interpretations each of which can be implemented as a program. Two such programs are given below as Java methods. In this simple implementation, parts are identified by an integer part number, both the cost in cents and the mark-up per cent, of parts, are stored in an integer array part cost mark-up[][]. These programs, and the system of which they are part, are not intended to illustrate high quality systems design. They are intended to illustrate the difficulty in identifying links between such imperative interpretations of chunks of system knowledge. The first method returns the sale price of an item given its part no:

```
static int part sale price( int part no, int part cost
           markup[][], int no of part nos ) {
 if (no of part nos < 1 ) return -1;
 for ( int count = 0; count < no of part nos; count++ ){
   if ( part cost markup[count][0] == part no ) {
     return part cost markup[count][1] *
        part cost markup[count][2] / 100;
   }
 }
 return -1;
} [P1]
```

The second method returns the part number of a part that sells for a given sale price:

```
static int sale price part( int sale price, int part cost
           markup[][], int no of part nos ) {
  if (no of part nos < 1 ) { return -1; }
  for ( int count = 0; count < no of part nos; count++ ) {
    if (( part cost markup[count][1]*part cost
        markup[count][2]/ 100 ) == sale price ) {
      return part cost markup[count][0];
```

```
  }
}return -1;
} [P2]
```

These two methods are related in that they are both based on different imperative interpretations of the same knowledge chunk. So if it is necessary to modify one of them then it may be necessary to modify the other. Further these two methods can not necessarily be guaranteed to reside in the same object in the system design although they are clearly both intimately associated with the array part cost markup[][]. Informally, the second is similar to the first but with "the implication going the other way". From a maintenance perspective, it is reasonable to estimate that the possibility of the relationship between [P1] and [P2] being overlooked is neither 'high' nor 'insignificant'.

Chunk [K2] also has a number of different imperative interpretations. One such interpretation may be used to calculate the profit on an item given its part no:

```
static int part profit( int part no, int part cost
           markup[][], int no of part nos ) {
  if (no of part nos < 1 ) { return -1; }
  for ( int count = 0; count < no of part nos; count++ ) {
    if ( part cost markup[count][0] == part no ) {
      return ( part cost markup[count][1] *
    ( part cost markup[count][2] - 100 ) ) / 100 ;
  }
}return -1;
} [P3]
```

Method [P3] is closely related to both methods [P1] and [P2] but this relationship is more obscure than the relationship between [P1] and [P2]. To make maters worse, [P3] is also related to chunk [K3] which may not have even been identified. So the possibility of the relationship between [P3] and {[P1], [P2]} being overlooked is high, and between [P3] and [K3] is very high — after all, [K3] has yet to be identified! Further the system knowledge embedded in [P3] may also manifest itself in a method to calculate the profit on other things besides 'parts' whose data is not stored in a two-dimensional array. Such a method should also be linked to the above. The simplicity of these examples should not detract from the importance of the principle that the failure to identify relationships between the chunks of knowledge, imperative interpretations of which are high level descriptions of methods,

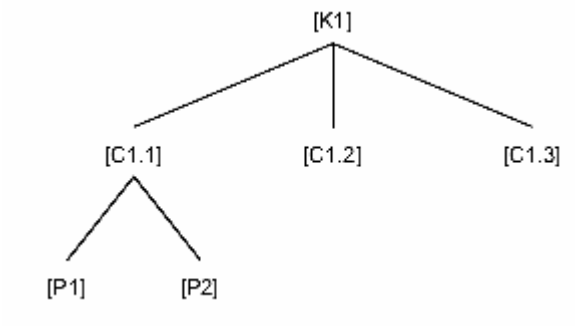
leads to maintenance hazards. The analysis presented here removes these hazards completely.

2.2. Declarative representations

In a declarative representation an if-then interpretation of a knowledge chunk is represented directly in an "if-then" formalism. In the 1980s there was considerable interest in building expert systems. At that time declarative formalisms, in particular if-then formalisms such as logic programming, provided one way of computing with knowledge that was far easier to use than imperative formalisms. The comparative ease of use of ifthen formalisms was responsible for the misapprehension that knowledge could be thought of as "if-then stuff". In a sense this is true. If a chunk of knowledge has a number of if-then interpretations then it is unlikely that more than one of those interpretations will be useful at any particular time. One consequence of this misapprehension is that changes in the validity of one if-then interpretation that is *not* implemented may have subtle implications for the validity of a number of parts of the knowledge base that *are* implemented. Approaches to modelling expert systems applications were also based on other than declarative representations; for example, on frame-based systems, but these are not considered here. During the 'age of expert systems' it was not uncommon to hear knowledge engineers observe: "I took considerable trouble to build the knowledge base well but now I find that a simple change in the application can lead to an extensive maintenance task". One reason for such an observation is that apparently useless links in the raw knowledge have been ignored.

In the hype that surrounded the early days of Prolog, the declarative paradigm appeared to offer significant benefits to the representation and maintenance of system knowledge. For example, a sort program in Prolog if 'driven backwards' can be used as a — not necessarily efficient — permutation generator. But at least as far as the representation of knowledge is concerned, declarative formalisms enable different imperative interpretations to be bundled into one declarative program. So — in theory — declarative formalisms reduce the number of possible program interpretations of system knowledge, and so — in theory — assist with maintenance. One problem *in practice* is that, as with the sort program

Fig 1. The abstraction hierarchy for chunk [K1]: knowledge rules and programs



mentioned above, it is difficult to write programs in Horn clause logic, or in any declarative formalism, that operate efficiently in both forward and reverse gear. Setting this issue aside for the moment, Horn clause logic provides a clean representation of system knowledge, but it is not powerful enough to represent a single chunk of system knowledge in a single logic program.

Consider the chunk of knowledge [K1] as given above. This single chunk is a simple statement of fact: it is not in an if-then form. Under a reasonable understanding of the meaning of chunk [K1] it admits three different if-then interpretations:

part/sale-price(x,y) ← *part/cost-price(x, z),*
part/mark-up(x,w),y = (z ·w) [C1.1]

part/cost-price(x, z) ← *part/sale-price(x,y),*
part/mark-up(x,w),y = (z ·w) [C1.2]

part/mark-up(x,w) ← *part/sale-price(x,y),*
part/cost-price(x, z),y = (z ·w) [C1.3]

For the third of these if-then interpretations — with "*part/mark-up*" as its head — there is a possibility of round-off error. Each of these three clauses may be driven in two directions, alternatively, using the powerful string-matching 'unification' method of Prolog they may pass partly assembled data structures in and out as arguments. For example, [C1.1] may be used both to find the sale price of a given part, and to find a part with a given sale price. Despite this representation power, all three clauses are required to capture

all of the wisdom in chunk [K1]. The three clauses [C1.1] — [C1.3] are an inconvenient representation of the single chunk [K1] in that one statement of fact has been represented as three logical statements. A hierarchy is emerging in these examples. In it a chunk of knowledge is interpreted as a set of if-then statements eachof which is interpreted as a set of imperative programs. The hierarchy for chunk [K1] is shown in Fig. 1.

Consider the chunk of knowledge [K2] as given above. As for [K1], the chunk [K2] is not in an if-then form. Under a reasonable understanding of its meaning, it also admits three if-then interpretations:

$$\textit{part/profit(x,y)} \leftarrow \textit{part/cost-price(x,w), part/mark-up(x,u),}\ z = (w \times u), y = z - w \quad [C2.1]$$

$$\textit{part/cost-price(x,w)} \leftarrow \textit{part/profit(x,y), part/mark-up(x,u),}\ z = (w \times u), y = z - w \quad [C2.2]$$

$$\textit{part/mark-up(x,u)} \leftarrow \textit{part/cost-price(x,w), part/profit(x,y),}\ z = (w \times u), y = z - w \quad [C2.3]$$

The six Horn clauses above [C1.1] — [C2.3] may be combined using resolution to give some potentially useful clauses:

$$\textit{part/profit(x,y)} \leftarrow \textit{part/cost-price(x,w), part/mark-up(x,u),}\ z = (w \times u), y = z - w \quad [C3]$$

$$\textit{part/profit(x,y)} \leftarrow \textit{part/sale-price(x, z), part/mark-up(x,u),}\ z = (w \times u), y = z - w \quad [C4]$$

$$\textit{part/mark-up(x,w)} \leftarrow \textit{part/sale-price(x,y), part/profit(x,u),}\ u = y - z, y = (z \times w) \quad [C5]$$

as well as some rather useless clauses:

$$\textit{part/sale-price(x,y)} \leftarrow \textit{part/sale-price(x,v), part/profit(x,u),}\ u = v - z, \textit{part/mark-up(x,w)}, y = (z \times w) \quad [C6]$$

A danger with all of [C3]—[C6] is that they are assembled from particular if-then interpretations of two chunks of knowledge, namely [K1] and [K2]. If the meaning of either of those two chunks should change then [C3] — [C6] may all have to be changed as well. This is not a problem if the relationships between [C3] — [C6] and both [K1] and [K2] are represented. But in the declarative representation, it is not clear that the meanings of [K1] and [K2] are buried in the four clauses [C3] — [C6]. So [C3] — [C6] and any other clauses derived from the original six clauses above, are potential maintenance hazards [10]. Given the six original clauses [C1.1] —[C1.3],

[C3.1] — [C3.3] there is no reason to combine them as illustrated in [C3] — [C6]. But clauses such as [C3] — [C6] are valid and could have been constructed by a programmer as part of a system. If they form part of a system then they may constitute a maintenance hazard [10]. To make matters worse, chunk [K3] is buried inside the three clauses [C2.1] — C[2.3].

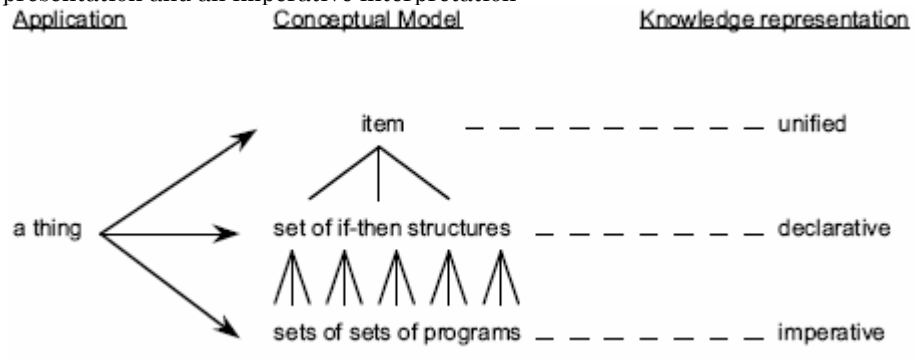
The discussion in this sub-section and in the previous sub-section is not intended to imply that traditional programming languages and methods should be discarded. The point is that no matter what programming language is used the problem of maintaining programs in that language is ideally tackled through a high level model of knowledge with the property that one chunk of real knowledge is represented in one place and for which there is a method for representing and removing the relationships between those chunks [11]. Such a model is described below. If the chunks represented in this model are linked to the programs that implement them then these links together with those in the model itself provide a maintenance map for the programs.

3. A unified representation

A representation formalism is a *unified representation* if all "data", information" and "knowledge" things are represented in the same way. The terms data, information and knowledge are used here in the following sense. The *data* things in an application are the fundamental, indivisible things. Data things can be represented as simple constants or variables. If an association between things *cannot* be defined as a succinct, computable statement then it is an *implicit* association. Otherwise it is an *explicit* association. An *information* thing in an application is an implicit association between data things. Information things can be represented as tuples or relations. A *knowledge* thing in an application is an explicit association between information and/or data things. Knowledge can be interpreted either as programs in an imperative language or as rules in a declarative language.

The expressive power of a unified representation must be able to describe *at least* the data, information and knowledge things. The unified

Fig.2. A thing and its representation in the unified representation, a declarative representation and an imperative interpretation



representation described here also contains two classes of constraints that apply equally to knowledge, information and to data. These constraints provide safeguards against invalid maintenance operations. In [12] these constraints are generalised to fuzzy acceptability measures of knowledge base integrity. Item and object join has been extended to apply to those measures [10].

Why use a unified knowledge representation [2]? A knowledge representation with the property that a single chunk of system knowledge is represented as a single entity is at a level of abstraction that is far closer to 'reality' than traditional declarative or imperative formalisms. There is a hierarchy: a real chunk of knowledge is represented as a single "item" in the unified representation. Each item has a number of interpretations as if-then forms. Each if-then form has a number of interpretations as imperative programs. This is illustrated in Fig. 2.

Further, if the unified knowledge representation treats data, information and knowledge in the same way then links between these three classes of things may also be represented. The majority of knowledge representation formalisms treat these three classes quite differently and so such links have no natural representation. Items represent *all* data, information and knowledge things in an application [4].

Items incorporate two powerful classes of constraints. The key to this uniform representation is the way in which the "meaning" of an item, called its semantics, is specified. The *semantics* of an item is a function that *recognises* the members of the "value set" of that item. The

part/sale-price		*part/cost-price*		*part/mark-up*	
1234	1.45	1234	1.20	1234	1.2
2468	2.99	2468	2.30	2468	1.3
3579	4.14	3579	3.45	3579	1.2
1357	10.35	1357	4.50	1357	2.3
9753	12.06	9753	6.70	9753	1.8
8642	12.78	8642	8.52	8642	1.5
4321	5.67	4321	2.70	4321	2.1

Table 1. Value set for the knowledge item [*part/sale-price, part/cost-price, part/mark-up].*

value set of an item will change in time *t*, but the item's semantics should remain constant. The value set of a data item at a certain time *t* is the set of labels that are associated with a population that implements that item at that time. The value set of an information item at a certain time *t* is the set of tuples that are associated with a relational implementation of that item at that time. Knowledge items have value sets too. Consider again the chunk of knowledge [K1] "the sale price of a part is the cost price of that part marked up by the mark-up rate for that part"; this chunk is represented by the item named *[part/sale-price, part/cost-price, part/markup]* with a value set of corresponding sextuples. This example illustrates a preference for using binary relations. When system knowledge is expressed in terms of such binary relations it tends to be in a simple form. A possible value set for this chunk is shown in Table 1. This chunk admits three interpretations as declarative rules which in turn lead to at least five non-trivial imperative programs.

The idea of defining the semantics of items as recognising functions for the members of their value set extends to complex, recursive knowledge items. Consider the chunk of knowledge "If two persons have the same address then they are cohabitants". This chunk can be represented by the item: *[person/cohabitant, person/address]*. The meaning of this item may be defined by the single clause:

$$\text{person/cohabitant}(x,y) \leftarrow \text{person/address}(x, z), \; \textit{person/address}(y, z), x \neq y$$

The semantics of an item is the recognising function for its value set. The trick to dealing with recursive items is to identify the correct value set for this purpose. A first attempt at constructing the value set of this chunk could be the four-tuples associated with the two information items *person/cohabitant* and *person/address*, but it is hard to construct a

recognising function for this value set. A function may be specified that recognises the value set of this item displayed in a different way. Consider the function:

$$\begin{aligned}(u,v,w,x,y,z) \in \text{ Value set of [person/cohabitant, person/address]} \leftrightarrow \\ (u,v) \in \text{Value set of } \textit{person/cohabitant} \wedge \\ (w,x) \in \text{Value set of } \textit{person/address} \wedge \\ (y,z) \in \text{Value set of } \textit{person/address} \wedge \\ (((u = w) \wedge (v = y) \wedge (u \neq v)) \wedge (x = z))\end{aligned}$$

This function recognises the tuples in the value set consisting of the three information items *person/cohabitant*, *person/address* and *person/address*. Hence if this set of six-tupes is the value set then the *[person/cohabitant, person/address]* item has a simple recognising function. The trick here is to use a double occurrence of the component information item *person/address*. The semantics of recursive knowledge items are defined in this way with value sets that have multiple occurrences of component information items.

An *item* is a named triple $A[SA, VA, CA]$ with *item name* A, SA is the *item semantics* of A, VA is the *item value constraints* of A and CA is the *item set constraints* of A. The item semantics, SA, is a λ-calculus expression that recognises the members of the value set of item A. The expression for an item's semantics may contain the semantics of other items A1,...,An which are called that item's *components*.

$$\begin{aligned}\lambda_{X1X2Y1Y2Z1Z2} \cdot [(&S_{\text{part/sale-price}}(x1,x2) \wedge S_{\text{part/cost-price}}(y1,y2) \wedge \\ &S_{\text{part/mark-up}}(z1, z2)) \wedge \\ &(((x1 = y1) \wedge (x1 = z1)) \rightarrow (x2 = z2 \times y2))] \cdot\end{aligned}$$

This expression asserts that the pair $(x1,x2)$ satisfies the semantics of the item *part/sale-price*, that $(y1,y2)$ satisfies the semantics of the item *part/cost-price*, that $(z1, z2)$ satisfies the semantics of the item *part/markup*, and that:

$$(((x1 = y1) \wedge (x1 = z1)) \rightarrow (x2 = z2 \times y2))$$

holds. This last component of the semantics expression is the substantive part. The essence of this expression is represented in the 'schema notation' in

Table 2 — the meaning of the last two rows in that Figure are discussed below.

item name	*[part/sale-price, part/cost-price, part/mark-up]*		
components	*part/sale-price*	*part/cost-price*	*part/mark-up*
dummy variables	(x,w)	(x,y)	(x,z)
semantics	$\rightarrow (w = z \times y)$		
constraints	$\rightarrow (w > y)$		
set constraints	$\emptyset$		

Table 2. Shema for the item [*part/sale-price, part/cost-price, part/mark-up].*

The schema notation is intended to make the unified representation more accessible. The important feature of the λ-calculus form or the schema form is that they each represent all that chunk [K1] says. For example, the two programs [P1] and [P2] are both interpretations of the item whose semantics is shown in Table. 2.

In general the item semantics is an expression of the form:

$$\lambda y_1^1 \ldots y_m^1 \ldots y_{m_n}^n \cdot [S_{A_1}(y_1^1, \ldots, y_{m_1}^1) \wedge \cdots \wedge [S_{A_n}(y_1^n, \ldots, y_{m_n}^n) \quad \wedge \\ J(y_1^1, \ldots, y_m^1, \ldots, y_{m_n}^n)] \cdot$$

where J is a first-order predicate. The item value constraints, VA, is a λ-calculus expression:

$$\lambda y_1^1 \ldots y_m^1 \ldots y_{m_n}^n \cdot [V_{A_1}(y_1^1, \ldots, y_{m_1}^1) \wedge \cdots \wedge [V_{A_n}(y_1^n, \ldots, y_{m_n}^n) \quad \wedge \\ K(y_1^1, \ldots, y_m^1, \ldots, y_{m_n}^n)] \cdot$$

where K is a first-order predicate, that should be satisfied by the members of the value set of item A as they change in time. So if a tuple satisfies SA then it should satisfy VA [13]. The expression for an item's value constraints contains the value constraints of that item's components. The item set constraints, CA, is an expression of the form:

$$C_{A_1} \wedge C_{A_2} \wedge \cdots \wedge C_{A_n} \wedge (L)_A$$

where L is a logical combination of:
– Card lies in some numerical range;
– Uni(Ai) for some i, $1 \leq i \leq n$, and
– Can(Ai,X) for some i, $1 \leq i \leq n$, where X is a non-empty subset of

$\{A1, \ldots, An\} - \{Ai\}$

subscripted with the name of the item A, "Uni(a)" means that "all members of the value set of item a must be in this association". "Can(A,X)" means that "the value set of the set of items X functionally determines the value set of item A". "Card" means "the number of things in the value set". The subscripts indicate the item's components to which that set constraint applies.

For example, each part may be associated with a cost-price subject to the "value constraint" that parts whose part-number is less that 1999 should be associated with a cost price of no more than $300. A set constraint specifies that every part must be in this association, and that each part is associated with a unique cost-price. The information item named *part/cost-price* then is:

$$\begin{aligned} \textit{part/cost-price}[&\lambda xy \cdot [S_{part}(x) \wedge S_{cost\text{-}price}(y) \wedge \text{costs}(x,y)]\cdot, \\ &\lambda xy \cdot [V_{part}(x) \wedge V_{cost\text{-}price}(y) \wedge ((x < 1999) \rightarrow (y \leq 300))]\cdot, \\ &C_{part} \wedge C_{cost\text{-}price} \wedge \\ &(\text{Uni}(part) \wedge \text{Can}(cost\text{-}price, \{part\})_{part/cost\text{-}price}] \end{aligned}$$

Chunks of knowledge can also be defined as items, although it is neater to define knowledge items using "objects", see Sec 3.1. "Objects" are item building operators. The knowledge item *[part/sale-price, part/cost-price, mark-up]* which means "the sale price of parts is the cost price marked up by a uniform markup factor" is:

$$\begin{aligned} &\textit{[part/sale-price,part/cost-price, part/mark-up]}[\\ &\lambda x_1 x_2 y_1 y_2 z_1 z_2 \cdot [(S_{part/sale\text{-}price}(x_1,x_2) \wedge S_{part/cost\text{-}price}(y_1,y_2) \wedge \\ &S_{part/mark\text{-}up}(z_1,z_2)) \wedge (((x_1 = y_1) \wedge (x_1 = z_1)) \rightarrow (x_2 = z_2 \times y_2))], \\ &\lambda x_1 x_2 y_1 y_2 z_1 z_2 \cdot [(V_{part/sale\text{-}price}(x_1,x_2) \wedge V_{part/cost\text{-}price}(y_1,y_2) \wedge \\ &V_{part/mark\text{-}up}(z_1,z_2)) \wedge ((x_1 = y_1) \rightarrow (x_2 > y_2))]\cdot, \\ &C_{[part/sale\text{-}price,part/cost\text{-}price,\ mark\text{-}up]}] \end{aligned}$$

The λ-calculus representation is rather clumsy. In practice items are represented in a more convenient schema notation. The schema notation for the above knowledge item, including its constraints, is illustrated in Table 2.

Two different items can share common embedded knowledge. This is a generalisation of the particular point made above that two programs may

implement the essence of the 'mark-up knowledge' in different contexts besides the mark-up of 'spare parts'. If this is so then those two items constitute a maintenance hazard. This problem can be avoided to some extent by using objects. If two items share some common embedded knowledge then declarative rules and imperative programs derived from them may also share common knowledge. For this reason, basing the approach to knowledge base maintenance at the abstract level of items simplifies maintenance.

3.1. Objects

To make the inherent structure of items clear, 'objects' are introduced as item building operators. The use of objects to build items enables the hidden links in the knowledge to be identified. A single operation for objects enables these hidden links to be removed from the knowledge thus simplifying maintenance.

An *n*-adic *object* is an operator that maps *n* given items into another item for some value of *n*. Further, the definition of each object will presume that the set of items to which that object may be applied are of a specific "type". Examples of item *type* include D^m for *m*-adic data items, I^m for *m*-adic information items and K^m for m-adic knowledge items. Items may also have unspecified, or free, type which is denoted by X^m. The formal definition of an object is similar to that of an item. An *object* named **A** is a typed triple $\mathbf{A}[E,F,G]$ where E is a typed expression called the *semantics* of A, F is a typed expression called the *value constraints* of A, and G is a typed expression called the *set constraints* of A. For example, the *part/cost-price* item can be built from the items *part* and *cost-price* using the **costs** operator:

$$\textit{part/cost-price} = \mathbf{costs}(\textit{part}, \textit{cost-price})$$

$$\mathbf{costs}[\lambda P:\mathrm{X}^1 Q:\mathrm{X}^1 \cdot \lambda xy \cdot [S_P(x) \wedge S_Q(y) \wedge \textit{costs}(x,y)] \cdot\cdot,$$
$$\lambda P:\mathrm{X}^1 Q:\mathrm{X}^1 \cdot \lambda xy \cdot [V_P(x) \wedge V_Q(y) \wedge ((1000 < x < 1999) \rightarrow (y \leq 300))] \cdot\cdot,$$
$$\lambda P:\mathrm{X}^1 Q:\mathrm{X}^1 \cdot [C_P \wedge C_Q \wedge (\mathrm{Uni}(P) \wedge \mathrm{Can}(Q,P))_{\nu(\mathbf{costs},P,Q)} \cdot]$$

where $\nu(\mathbf{costs},P,Q)$ is the name of the item $\mathbf{costs}(P,Q)$. As for items, objects are more digestible in the schema notation. The schema for the **costs** object is

shown in Table 3 where universal set constraints are denoted by an '$\forall$' and candidate constraints by an '$\otimes$' and a '——'.

object name	costs	
argument type	X^1	X^1
dummy variables	(x)	(y)
semantics	costs(x,y)	
value constraints	$x < 1999 \rightarrow y \leq 300$	
set constraints	$\forall$	
	——	$\odot$

Table 3. The shema for the object costs

Data objects provide a representation of sub-typing. Data objects are used in the conceptual model to derive individual data items from the universal item U, where $U = U[\lambda x \cdot x : \mathrm{U} \cdot, \lambda x \cdot > \cdot, /0]$, "U" is the "universe of discourse" and > is the constant "true" expression. The data object **part** is:

$$\textbf{part}[\lambda P : \mathrm{X}^1 \cdot \lambda x \cdot [S_P(x) \wedge \text{with-part}(x)] \cdot \cdot,$$
$$\lambda P : \mathrm{X}^1 \cdot \lambda x \cdot [V_P(x) \wedge (x \geq 0)] \cdot \cdot, \lambda P : \mathrm{X}^1 \cdot [C_P \wedge (\mathrm{Card} \geq 1)_{\mathrm{v}(part,P)}] \cdot]$$

If the object **part** is applied to the universal item U it then generates the item *part* by: *part* = **part***U*.

Declarative rules are quite clumsy when represented as items; objects provide a more compact representation. For example, consider the *[part/sale-price, part/cost-price, part/mark-up]* knowledge item which represents the chunk of knowledge [K1] "The sale price of a part is the cost price of that part marked up by the markup rate for that part". This item can be built by applying a knowledge object **mark-up-rule** of argument type (I^2, I^2, I^2) to the items *part/sale-price*, *part/cost-price* and *part/markup*. That is:

[part/sale-price,part/cost-price, part/mark-up] =
mark-up-rule(*part/sale-price, part/cost-price, part/mark-up)*

mark-up-rule		
X^2	X^2	X^2
(x,w)	(x,y)	(x,z)
$\rightarrow (w = z \times y)$		
$\rightarrow (w > y)$		
$\forall$	$\forall$	
——		$\odot$
$\odot$	——	
——	$\odot$	——

Table 4. The shema for the object **mark-up-rule**

Objects also represent value constraints and set constraints in a uniform way. The **mark-up-rule** object is:

$$\begin{aligned}
&\textbf{mark-up-rule}[\lambda P : \mathrm{X}^2 Q : \mathrm{X}^2 R : \mathrm{X}^2 \cdot \lambda x_1 x_2 y_1 y_2 z_1 z_2 \cdot [S_P(x_1,x_2) \wedge \\
&S_Q(y_1,y_2) \wedge S_R(z_1,z_2) \wedge (((x_1 = y_1) \wedge (x_1 = z_1)) \rightarrow (x_2 = z_2 \times y_2))] \cdot \cdot, \\
&\lambda P : \mathrm{X}^2 Q : \mathrm{X}^2 R : \mathrm{X}^2 \cdot \lambda x_1 x_2 y_1 y_2 z_1 z_2 \cdot [V_P(x_1,x_2) \wedge V_Q(y_1,y_2) \wedge \\
&V_R(z_1,z_2) \wedge ((x_1 = y_1) \rightarrow (x_2 > y_2))] \cdot \cdot, \\
&\lambda P : \mathrm{X}^2 Q : \mathrm{X}^2 R : \mathrm{X}^2 \cdot [C_P \wedge C_Q \wedge C_R \wedge (\mathrm{Uni}(P) \wedge \mathrm{Uni}(Q) \wedge \\
&\mathrm{Can}(P,\{Q,R\}) \wedge \mathrm{Can}(Q,\{P,R\}) \wedge \mathrm{Can}(R,\{P,Q\}))_{\mathrm{v(mark\text{-}up\text{-}rule},P,Q,R)}] \cdot]
\end{aligned}$$

and its schema form is shown in Table 4.

3.2. The join operator

Item join provides the basis for item decomposition [4]. Given items A and B, the item with name $AE\,B$ is called the *join* of A and B on E, where E is a set of components common to both A and B. Consider:

$$A[S_A, V_A, C_A] \quad \text{and} \quad B[S_B, V_B, C_B]$$

Suppose that S_A has n variables, that is A is an n-adic item. Suppose that S_B has m variables, that is B is an m-adic item. Some of the components of A and B may be identical. Suppose that k pairs of components of A and B that are identical are identified, where $k \geq 0$. Let E be an ordered set of components where each is one of these identical pairs of components of both A and B. E

may be empty. To ensure that the definition is well defined the order of the components in the set E is the same as order in which they occur as components of A. Suppose the semantics expressions of the components from item A (or item B) that are in the set E are expressed in terms of a total of p variables. Let A^* be an n-adic item that is identical to item A except for the order of its variables. The last p variables in A^* are those variables in A that belong to the components of A in the set E. Let B^* be an m-adic item that is identical to item B except for the order of its variables. The first p variables in B^* are those variables in B that belong to the components of B in the set E. Let π' be a permutation that turns the ordered set of variables of A^* into the ordered set of variables of A. Let π be a permutation that turns the ordered set of variables of B^* into the ordered set of variables of B. Suppose that x is an $(n-p)$-tuple of free variables, y is a p-tuple of free variables and z is an $(m-p)$-tuple of free variables. Then the item with name $A \otimes_E B$ is the *join* of A and B on E, it is defined to be:

$$(A \otimes_E B)[\lambda xyz \cdot [S_A(\pi(x,y)) \wedge S_B(\pi'(y,z))]\cdot,$$
$$\lambda xyz[V_A(\pi(x,y)) \wedge V_B(\pi'(y,z))]\cdot, C_{A \otimes_E B}]$$

where $C_A \otimes_{EB}$ is defined as follows. Suppose that C_A is an expression of the form $c_A \wedge G$ where c is that part of C_A that carries the subscript 'A' and G is that part of C_A that carries subscripts other than 'A'. Likewise suppose that C_B is an expression of the form $d_B \wedge H$. Then:

$$C_{A \otimes_E B} \triangleq (c \wedge d)_{A \otimes_E B} \wedge (G \wedge H)$$

The set E is a set of identical pairs of components of A and B. If E is the set of all identical pairs of components of A and $_B$ then $A \otimes_E B$ may be written as $A \otimes B$.

Using the method of composition , knowledge items, information items and data items may be joined with one another regardless of type. For example, the knowledge item:

$$[\text{cost-price, tax}][\lambda xy \cdot [S_{\text{cost-price}}(x) \wedge S_{\text{tax}}(y) \wedge x = y \times 0.05]\cdot,$$
$$\lambda xy \cdot [V_{\text{cost-price}}(x) \wedge V_{\text{tax}}(y) \wedge x < y]\cdot, C_{[\text{cost-price, tax}]}]$$

can be joined with the information item *part/cost-price* on the set {*costprice*} to give the information item *part/cost-price/tax*. In other words:

$$
\begin{aligned}
&[\text{cost-price, tax}] \otimes_{\{\text{cost-price}\}} \text{part/cost-price} = \\
&\text{part/cost-price/tax}[\lambda xyz \cdot [S_{\text{part}}(x) \wedge S_{\text{cost-price}}(x) \wedge S_{\text{tax}}(y) \wedge \\
&\quad \text{costs}(x,y) \wedge z = y \times 0.05]\cdot, \\
&\lambda xyz \cdot [V_{\text{part}}(x) \wedge V_{\text{cost-price}}(x) \wedge V_{\text{tax}}(y) \wedge \\
&\quad ((1000 < x < 1999) \rightarrow (0 < y \leq 300)) \wedge (z < y)]\cdot, C_{\text{part/cost-price/tax}}]
\end{aligned}
$$

Using the item join operator, items may be joined together to form more complex items. The operator also forms the basis of a theory of decomposition in which each item is replaced by a set of simpler items. An item I is *decomposable*; into the set of items $D = \{I_1, I_2, ..., I_n\}$ if: I_i has non-trivial semantics for all i, $I = I_1 \otimes I_2 \otimes \cdots \otimes I_n$, where each join is *monotonic*; that is, each term in this composition contributes at least one component to I. If item I is decomposable then it will not necessarily have a unique decomposition. The join operator for objects is defined in a similar way to item join and is also denoted by . When is used to join two objects it is subscripted with a set of pairs of positive integers that identify the component pairs in the first and second argument ofthat are being joined. The method of decomposition is: "Given a conceptual model discard any items and objects which are decomposable". For example, this method requires that the item *part/cost-price/tax* should be discarded in favour of the two items *[cost-price,tax]* and *part/cost-price*.

4. Analysis of the conceptual model

A conceptual model consists of:
– the universal data item U,
– an object library,
– a conceptual diagram, and
– a set of maintenance links.

where, U, the universal data item, is as defined above. The *conceptual diagram* is a graph in which each item is represented by a node and is linked to those nodes from which it can be derived by applying an object operator. The conceptual diagram also shows the programs that implement the model

and links them to the knowledge-items from which they are derived. So the items in the conceptual model are constructed by applying a set of object operators to U. A *maintenance link* joins two items in the conceptual model if modification of one item means that the other item must be checked for correctness, and maybe modified, so that the consistency of the conceptual model is preserved [14]. The number of maintenance links can be very large. So maintenance links can only form the basis of a practical approach to knowledge base maintenance if there is some way of reducing their density on the conceptual model.

RESULT.
Sub-item links may be reduced to sub-type links between data items.
Demonstration:
Given two items A and B, where both are n-adic items with semantics S_A and S_B respectively, if π is permutation such that:

$$(\forall x_1 x_2 \ldots x_n)[S_A(x_1, x_2, \ldots, x_n) \leftarrow S_B(\pi(x_1, x_2, \ldots, x_n))]$$

then item B is a *sub-item* of item A. These two items should be joined with a maintenance link. If A and B are both data items then B is a sub-type of A. Suppose that:

$$X = \mathbf{E}D; \text{ where } D = \mathbf{C}AB \qquad (1)$$

for items X, D, A and B and objects $\mathbf{E}$ and $\mathbf{C}$. Item X is a sub-item of item D. Object $\mathbf{E}$ has the effect of extracting a sub-set of the value set of item D to form the value set of item X. Item D is formed from items A and B using object $\mathbf{C}$. Introduce two new objects $\mathbf{F}$ and $\mathbf{J}$. Suppose that object $\mathbf{F}$ when applied to item A extracts the same subset of item A's value set as $\mathbf{E}$ extracted from the "left-side" (ie. the "A-side") of D. Likewise $\mathbf{J}$ extracts the same subset of B's value set as $\mathbf{E}$ extracted from D. Then:

$$X = \mathbf{C}GK; \text{ where } G = \mathbf{F}A \text{ and } K = \mathbf{J}B \qquad (2)$$

so G is a sub-item of A, and K is a sub-item of B. The form (2) differs from (1) in that the sub-item maintenance links have been moved one layer closer to the data item layer, and object $\mathbf{C}$ has moved one layer away from the data item layer. Using this method repeatedly sub-item maintenance links between non-data items are reduced to sub-type links between data items. □

RESULT.
There are four kinds of maintenance link in a conceptual model built using the uniform knowledge representation.
Demonstration:
Consider two items A and B, and suppose that their semantics S_A and S_B have the form:

$$S_A = \lambda y_1^1 \ldots y_{m_1}^1 \ldots y_{m_p}^p \cdot [S_{A_1}(y_1^1, \ldots, y_{m_1}^1) \wedge \cdots \wedge S_{A_p}(y_1^p, \ldots, y_{m_p}^p) \wedge J(y_1^1, \ldots, y_{m_1}^1, \ldots, y_{m_p}^p)] \cdot$$
$$S_B = \lambda y_1^1 \ldots y_{n_1}^1 \ldots y_{n_q}^q \cdot [S_{B_1}(y_1^1, \ldots, y_{n_1}^1) \wedge \cdots \wedge S_{B_q}(y_1^q, \ldots, y_{n_q}^q) \wedge K(y_1^1, \ldots, y_{n_1}^1, \ldots, y_{n_q}^q)] \cdot$$

S_A contains (p+1) terms and S_B contains (q+1) terms. Let μ be a maximal sub-expression of $S_A \otimes_B$ such that:

$$\text{both} \quad S_A \rightarrow \mu \quad \text{and} \quad S_B \rightarrow \mu \tag{3}$$

where μ has the form:

$$\lambda y_1^1 \ldots y_{d_1}^1 \ldots y_{d_r}^r \cdot [S_{C_1}(y_1^1, \ldots, y_{d_1}^1) \wedge \cdots \wedge S_{C_r}(y_1^r, \ldots, y_{d_r}^r) \wedge L(y_1^1, \ldots, y_{d_1}^1, \ldots, y_{d_r}^r)] \cdot$$

If μ is empty, ie. 'false', then the semantics of A and B are independent. If μ is non-empty then the semantics of A and B have something in common and A and B should be joined with a maintenance link.

Now examine μ to see why A and B should be joined. If μ is non-empty then there are three cases. First, if:

$$S_A \leftrightarrow S_B \leftrightarrow \mu \tag{4}$$

then items A and B are equivalent and should be joined with an *equivalence link*. Second if (4) does not hold and:

$$\text{either} \quad S_A \leftrightarrow \mu \quad \text{r} \quad S_B \leftrightarrow \mu \tag{5}$$

then either A is a sub-item of B, or B is a sub-item of A and these two items should be joined with a *sub-item link*. Third, if (4) and (5) do not hold then if Δ is a minimal sub-expression of S_A such that $\Delta \rightarrow \mu$. Then:

$$\text{either } S_{A_i}(v_1^j, \ldots, v_{m_j}^j) \in \Delta \text{ or some } j \quad (6)$$
$$\text{or } J(y_1^1, \ldots, y_{m_j}^1, \ldots, y_{m_p}^p) \in \Delta \quad (7)$$

Both (6) and (7) may hold. If (6) holds then items A and B share one or more component items to which they should each be joined with a *component link*. If (7) holds then items A and B may be constructed with two object operators whose respective semantics are logically dependent. Suppose that item A was constructed by object operator $\mathbf{C}$ then the semantics of $\mathbf{C}$ will imply:

$$\Phi = \lambda Q_1 : X_1^{j_1} Q_2 : X_2^{j_2} \ldots Q_j : X_j^{j_j} \cdot \lambda y_1^1 \ldots y_{d_1}^1 \ldots y_{d_r}^r \cdot [S_{P_1}(y_1^1, \ldots, y_{d_1}^1) \wedge \ldots \wedge S_{P_r}(y_1^r, \ldots, y_{d_r}^r) \wedge L(y_1^1, \ldots, y_{d_1}^1, \ldots, y_{d_r}^r)].$$

where the Qi's take care of any possible duplication in the Pj's. Let $\mathbf{E}$ be the object $\mathbf{E}[\Phi,>, /0]$ then $\mathbf{C}$ is a sub-object of $\mathbf{E}$; that is, there exists a non-tautological object $\mathbf{F}$ such that:

$$\mathbf{C} \approx_{\Pi'} \mathbf{E} \otimes_M \mathbf{F} \quad (8)$$

for some set M and where the join is not necessarily monotonic. Items A and B are weakly equivalent, written $A \approx \pi' B$, if there exists a permutation π such that:

$$(\forall x_1 x_2 \ldots x_n)[S_A(x_1, x_2, \ldots, x_n) \leftrightarrow S_B(\pi(x_1, x_2, \ldots, x_n))]$$

where the x_i are the n_i variables associated with the i'th component of A. If A is a sub-item of B and if B is a sub-item of A then items A and B are weakly equivalent.

If (8) holds then the maintenance links are of three different kinds. If the join in (8) is monotonic then (8) states that $\mathbf{C}$ may be decomposed into $\mathbf{E}$ and

F. If the join in (8) is not monotonic then (8) states that either $\mathbf{C} \approx \pi' \mathbf{E}$ or $\mathbf{C} \approx \pi' \mathbf{F}$. So, if the join in (8) is not monotonic then either **E** will be weakly equivalent to **C**, or **C** will be a sub-object of **E**. □

It has been shown above that sub-item links between non-data items may be reduced to sub-type links between data items. So if:

- all equivalent objects have been removed by re-naming, and
- sub-item links between non-data items have been reduced to sub-type links between data items

then the maintenance links will be between nodes marked with:

- a data item that is a sub-type of the data item marked on another node, these are called the *sub-type links*;
- an item and the nodes marked with that item's components, these are called the *component links*, and
- an item constructed by a decomposable object and nodes constructed with that object's decomposition, these are called the *duplicate links*.

If the objects employed to construct the conceptual model have been decomposed then the only maintenance links remaining will be the sub-type links and the component links. The sub-type links and the component links cannot be removed from the conceptual model.

Unfortunately, decomposable objects, and so too duplicate links, may be hard to detect. Suppose that objects **A** and **B** are decomposable as follows:

$$\mathbf{A} \approx_W \mathbf{E} \otimes_M \mathbf{F} \qquad \mathbf{B} \approx_W \mathbf{E} \otimes_M \mathbf{G}$$

Then objects **A** and **B** should both be linked to object **E**. If the decompositions of **A** and **B** have not been identified then object **E** may not have been identified and the implicit link between objects **A** and **B** may not be identified.

4.1. Identifying decomposable objects

Four principles are given that identify potentially decomposable objects. The first of these principles relies on the notion of a "separable" predicate.

Principle 1: Given a predicate J of the form:

$$J(y_1^1, \ldots, y_{m_1}^1, y_1^2, \ldots, y_{m_2}^2, \ldots, y_1^n, \ldots, y_{m_n}^n)$$

Define the set $\{Y_1, Y_2, \ldots, Y_n\}$ by $Yi = \{y^i_1, \ldots, y^i_{mi}\}$. If J can be written in the form $J_1 \wedge J_2 \wedge \cdots \wedge J_m$ where each J_i is a predicate in terms of the set of variables X_i with:

- $X_i \subset Y_1 \cup Y_2 \cup \cdots \cup Y_n$, and
- for each Xi $(\exists j)$ such that X_i does *not* contain any of the variables in Y_j

then predicate J is *separable* into the partition $\{X_1, X_2, \ldots, X_m\}$.

If the predicate in an object's semantics is separable then investigate whether that object is decomposable into objects containing the argument sets identified by the separability of that predicate.

Principle 2. Given object **C**, if the objects **A** and **B** are not tautological, and the argument sets X, Y and Z all non-empty with:

$$C_{\mathbf{C}}[X \Longleftarrow Y, Y \Longleftarrow Z], \quad C_{\mathbf{A}}[X \Longleftarrow Y] \text{ and } C_{\mathbf{B}}[Y \Longleftarrow Z]$$

where $\Longleftarrow$ indicates functional dependency, then check whether:

$$\mathbf{C}(X,Y,Z) = \mathbf{A}(X,Y) \otimes_{\{(2,1)\}} \mathbf{B}(Y,Z)$$

If it does then discard object **C** in favour of objects **A** and **B**.

Principle 3. Given object **C**, if the objects **A** and **B** are not tautological, and the argument sets X, Y and W all non-empty and:

$$C_{\mathbf{C}}[X \Longleftarrow (Y,W)], \quad C_{\mathbf{A}}[X \Longleftarrow Y] \text{ and } \quad C_{\mathbf{B}}[\emptyset]$$

then check whether:

$$\mathbf{C}(X,Z) = \mathbf{A}(X,Y) \otimes_{\{(2,1)\}} \mathbf{B}(Y,W)$$

If it does then discard object **C** in favour of objects **A** and **B**.

Principle 4. Given object **C**, if the objects **A** and **B** are not tautological, and the argument sets V, W, X, Y and Z all non-empty with $W \supset V \cup X$ and:

$$C_{\mathbf{C}}[Z \Longleftarrow W], \quad C_{\mathbf{A}}[Z \Longleftarrow V, Y] \text{ and } \quad C_{\mathbf{B}}[Y \Longleftarrow X]$$

then check whether:

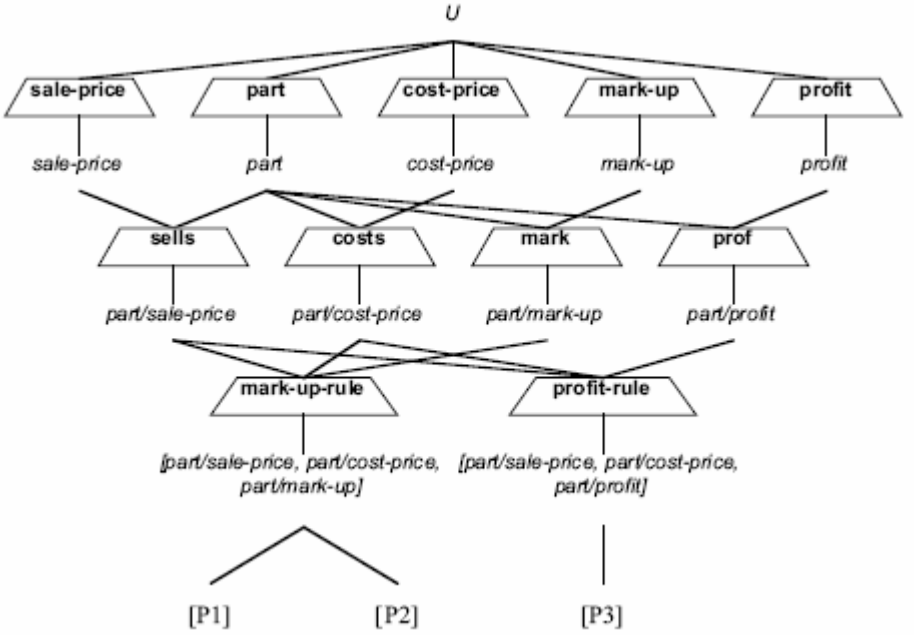
$$\mathbf{C}(Z,W) = \mathbf{A}(Y,Z,V) \otimes_{\{(1,1)\}} \mathbf{B}(Y,X)$$

If it does then discard object **C** in favour of objects **A** and **B**.

5. Maintaining the unified representation

How does the unified representation manage the maintenance of the imperative programs in Sec. 2.1 or the declarative programs in Sec. 2.2? Much of the formal paraphernalia to describe that example has been introduced in the preceding discussion. A conceptual diagram for the material described in Sec. ?? is shown in Fig. 3. That Figure shows the component

Fig. 3. Conceptual diagram for knowledge in example programs.

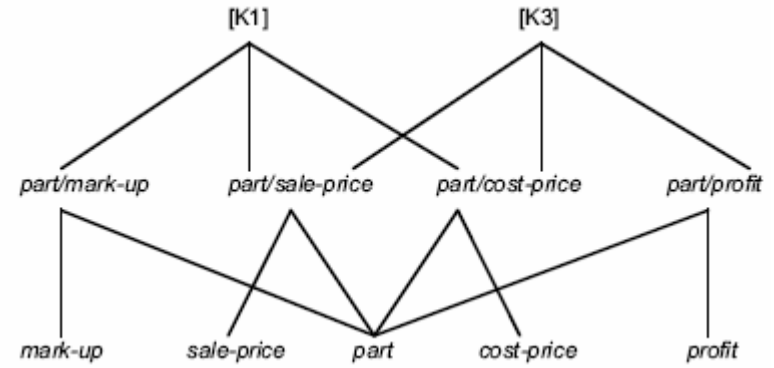


links. As all of the objects in that Figure are fully decomposed those links are also a complete set of maintenance links. Further, Fig. 3 does not show any duplicate use of objects — and this is part of the rationale for introducing objects—but that Figure is large enough as it is. It could have shown the mark-up-rule object being used to derive an item *[product/saleprice, product/cost-price, product/mark-up]* for example. The multiple use of objects in this way leads to a very intricate understanding of the maintenance structure.

To illustrate the use of this diagram suppose that there is some modification to the *mark-up* item. This modification could be a modification

to its constraints on mark-up values permitting mark-up values in a range that was previously not permitted. Will this modification require that the program [P3] requires maintenance? To answer this question follow all paths from **mark-up** to [P3] until an item is encountered that is not effected by this modification. The universal item *U* will never be effected by such a modification, for example. So first consider part, which, suppose is not effected. Second, suppose that *part/mark-up* is effected, but that the object **mark** is not. The only remaining

Fig. 4. Maintenance links for [K1] and [K2].



path from *part/mark-up* to [P3] leads through **mark-up-rule** to *part/cost-price*. Suppose that *part/cost-price* is not effected by the modification to *part/mark-up*. The chain then halts and so [P3] will not be effected by this modification.

As a simple example, the rule of decomposition is applied to reduce [K2] above to [K1] and [K3], so removing [K2] from the conceptual model. Maintenance links join two items in the conceptual model if modification of one of these items could require that the other item should be checked for correctness if the validity of the conceptual model is to be preserved. The efficiency of maintenance procedures depends on a method for reducing the density of the maintenance links in the conceptual model. One kind of maintenance link is removed by applying the rule of knowledge decomposition. Another is partly removed by reducing sub-item relationships to sub-type relationships. And another is removed by re-naming. In the simple example given the conceptual model consists only of [K1] and [K3], and the maintenance links are just the component links as shown in Fig. 4. So what? Because the model can not be decomposed it means that the maintenance links in Fig. 4 are *complete*. These links may then be mapped to

the imperative programs that implement the knowledge in chunks [K1] and [K3].

6. Constraints

The *conceptual model* consists of a representation of each thing as an item. Both items and objects contain two classes of constraint. These two classes are the value constraints and the set constraints. Constraints play a significant role in knowledge base maintenance. They are employed for two distinct purposes:

- constraints protect the validity of the knowledge base during maintenance [12] (these are called *pragmatic constraints*), and
- constraints contribute to the efficiency of the maintenance procedure (these are called referential constraints).

Pragmatic constraints are an integral part of every item and object in the conceptual model. Pragmatic constraints apply equally to knowledge, information and data. A taxonomy of pragmatic constraints is:

- constraints which are attached to each item (these are called the item constraints), these are:
  - the item value constraints which are constraints on the individual members of an item's value set, and
  - the item set constraints which are constraints on the structure of an item's value set. Set constraints include:
  - * cardinality constraints, denoted by "Card", which constrain the size of the value set;
  - * universal constraints, denoted by "Uni", which generalise database universal constraints, and
  - * candidate constraints, denoted by "Can", which constrain the functional dependencies in an item and generalise database key constraints.
  - * constraints which are attached to the conceptual model itself (these are called the model constraints).

The need to follow component links may be restricted by applying "referential constraints" to items. Referential constraints state that a particular component link need not be followed during the complete execution of a maintenance operation. They improve the efficiency of the maintenance procedure, but they complicate the maintenance of the item to which they are applied and so they should only be applied to items of low volatility. Model

constraints are constraints on the conceptual model. They are used in database technology. The rule "the selling price of parts is always greater than the cost price of parts" is an example of a chunk of knowledge that could be a constraint on the information in a database. The information in the database is constrained to be consistent with this particular chunk of knowledge. Such a constraint is a *knowledge model constraint*. They may be used for knowledge-based systems. For knowledgebased systems the inverse of this idea can be used. In knowledge-based systems a chunk of information can be used as a constraint on the knowledge in the conceptual model. Such a constraint is an *information model constraint*. Hand-coded, simple but non-trivial information models can provide powerful information model constraints. Information model constraints are simple, powerful and effective constraints on the knowledge in the conceptual model. They may be useful in applications where the knowledge is subject to a high rate of change and the information is comparatively stable.

7. Conclusion

A high-level abstraction of imperative programs is achieved by using a unified conceptual model. An item encapsulates the wisdom in a set of declarative rules each of which in turn encapsulates the wisdom in a set of imperative programs. Maintenance links form the basis of a maintenance procedure. A maintenance link joins two items in the conceptual model if a modification to one of them means that the other must necessarily be checked for correctness, and so possibly modified, if consistency of that set of items is to be preserved. The efficiency of this maintenance procedure depends on a method for reducing the density of the maintenance links. One kind of maintenance link may removed by applying a method of decomposition. Five principles have been given to identify decomposable objects. Another kind of maintenance link is removed by reducing subitem relationships to sub-type relationships. In this way the maintenance problem is simplified. All items, including knowledge items, have a set of constraints. The constraints of a knowledge item apply to any program that implements that item. So item constraints provide a mechanism that further protects against integrity violation during maintenance.

References

1. Kern-Isberner, G.: Postulates for conditional beief revision. In: proceedings International Joint Conference on Artificial Intelligence IJCAI'99, Stockholm, Sweden (1999) 186–191
2. Debenham, J.: Why use a unified knowledge representation? In: proceedings Fourteenth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-2001, Budapest, Hungary (2001)
3. Iglezakis, I., Reinartz, T., Roth-Berghofer, T.: Maintenance Memories: Beyond Concepts and Techniques for Case Base Maintenance. In: Advances in Case-Based Reasoning. Springer-Verlag (2004) 227 – 241
4. Debenham, J.: Knowledge Engineering — Unifying Knowledge Base and Database Design. Springer-Verlag (1998)
5. Mayol, E., Teniente, E.: Addressing the process of integrity maintenance. In: proceedings Tenth Intenational Conference on Database and Expert Systems DEXA'99, Florence, Italy (1999) 270 – 281
6. Katsuno, K., Mendelzon, A.: On the difference between updating a knowledge base and revising it. In: proceedings Second International Conference on Principles of Knowledge Representation and Reasoning KR'91, Morgan-Kaufmann (1991)
7. Barr, V.: Applying reliability engineering to expert systems. In: proceedings 12'th International FLAIRS conference. (1999) 494 – 498
8. Jantke, K., Herrmann, J.: Lattices of knowledge in intelligent systems validation. In: proceedings 12'th International FLAIRS conference, Florida, US (1999) 499 – 505
9. Darwiche, A.: Compiling knowledge into decomposable negation normal form. In: proceedings International Joint Conference on Artificial Intelligence IJCAI'99, Stockholm, Sweden (1999) 284 – 289
10. Debenham, J.: A rigorous approach to knowledge base maintenance. In: proceedings Sixteenth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-2003, Loughborough, UK (2003) 219 – 228
11. Roth-Berghofer, T.: Knowledge maintenace of case-based reasoning systems – the SIAM methodology. Zeitschrift KI – Kunstliche Intelligenz **17** (2003) 55 – 57
12. Debenham, J.: Fuzzy degrees of knowledge integrity. In: proceedings 9'th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems IPMU-2002. (2002) 1391 – 1398
13. Johnson, G., Santos, E.: Generalizing knowledge representation rules for acquiring and validating uncertain knowledge. In: proceedings 13'th International FLAIRS conference, Florida, US (2000) 186 – 191
14. Ramirez, J., de Antonio, A.: Semantic verification of rule-based systems with arithmetic constraints. In: proceedings 11'th International Conference on Database and Expert Systems DEXA'2000, London, UK (2000) 437 – 446

John Debenham is Professor of Computer Science at the University of Technology, Sydney. John is also Chair of the Australian Computer Society's National Committee for Artificial Intelligence and Expert Systems. John Debenham has a long standing research interest in the design of knowledge-based systems. During 1997 he developed an interest in multi-agent systems. He presently retains both of these interests. The focus of his research on knowledge-based systems has been on the preservation of system integrity.